Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Systém pro uložení grafické informace

Plzeň, 2010 Tomáš Hofhans

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 15. května 2010

Tomáš Hofhans

Abstract

System for storing graphics information

This diploma thesis contains two parts. First one is about creating an extensible system for storing images. The second part describes a new compression type for images based on the similarity of images. System for storing images and other files was designed and implemented. This system enables data storing and structuralization. The data can be stored in an authentic format as a link if the same file already exists in the system or the compression method described in the thesis can be used. Another compression methods can be simply integrated into the system. The new compression method is based on the JPEG method. Color transformation, block splitting, discrete cosine transform and quantization are the same. The compression method is looking for similarity in quantized blocks. The described method stores only unique or similar blocks. The compressed file is stored as references to these blocks. Unfortunately the researched method is not effective.

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

Práce řeší problematiku komprese obrazu. Přesněji řečeno se zabývá ztrátovou kompresí. Ze zadání úlohy vyplývá, že komprese by měla nějakým způsobem využívat podobnosti mezi jednotlivými obrazy nebo v obrazu samotném.

První polovina práce obsahuje teoretický úvod do problematiky. V této části se zabývám nejprve výčtem existujících typů komprese, na které je možné navázat a u kterých je možné získat určitou inspiraci pro nové řešení. Protože cílem je hledat podobnosti v obrazu nebo obrazech, je zapotřebí prostudovat metody dělení obrazu. Tuto tematiku popisuje kapitola Segmentace a dekompozice obrazu. Poslední část teoretického úvodu je věnována metodám porovnání obrazu. Tato kapitola je nepostradatelná hned ze dvou důvodů. Za prvé při hledání podobných částí je musíme nějakým způsobem porovnávat. Za druhé při ztrátové kompresi je vhodné provádět porovnání s originálním obrazem. Všechny tři teoretické kapitoly by měli čtenáři poskytnout základní informace o řešené problematice. Velká část popisovaných znalostí je použita v realizační části práce.

Druhá polovina obsahuje právě část realizační. Krom vytvoření kompresní metody bylo cílem práce vytvoření aplikace, která bude kompresní metodu poskytovat. První polovina realizační části je věnována technické dokumentaci implementované aplikace. Druhá polovina popisuje vývoj kompresní metody. Zkoumaná komprese vychází z velmi používané metody JPEG. Podobnost je aplikována na bloky obrazu, které jsou převedeny do frekvenční oblasti a kvantovány. Kromě podrobného popisu kompresní metody obsahuje kapitola Realizace komprese navíc výsledky měření, které mají dokázat popisovaná tvrzení.

Celá práce je zakončena závěrem, který shrnuje dosažené výsledky. Dále následují přílohy obsahující obrázky, diagramy, uživatelskou dokumentaci a podrobnosti k možnosti rozšíření implementované aplikace.

2 Teoretický úvod

2.1 Komprese dat

V této kapitole bylo čerpáno z [ŽBSF04, Wró04, W01] a některé uvedné příklady byly z těchto zdrojů převzaté.

2.1.1 Úvod

Kompresí dat rozumíme převod z jednoho datového formátu do jiného (neboli kódování dat) s cílem zmenšení datového objemu. Opačný převod do původního formátu se nazývá dekomprese.

Nejdůležitějším parametrem komprese je kompresní poměr. Jedná se o podíl velikosti původních dat a velikosti dat komprimovaných. Například pokud data o velikosti 12MiB zmenšíme kompresí na velikost 6MiB, dosáhli jsme kompresního poměru $12/6 = 2$ (tj. $2:1$ – dvě ku jedné, zmenšeno na polovinu).

Podle toho, zda se data originálu a dekomprimovaná data shodují, lze komprese rozdělit do dvou kategorií:

Bezztrátová komprese – data po kompresi a následné dekompresi jsou naprosto shodná s původními daty. Používá se v případě, kdy nesmí dojít ke ztrátě nebo poškození dat.

Ztrátová komprese – dochází k částečnému zjednodušení a tedy poškození komprimovaných dat. Původní data není možno zpětně rekonstruovat do původní podoby. V určitých aplikacích nám tento fakt nevadí, protože zkreslení je malé. Ztrátová komprese mívá zpravidla lepší kompresní poměr.

Samozřejmě existuje mnoho dalších parametrů a způsobů dělení komprese. Důležitá je rychlost komprese a dekomprese. Pro přehlednost a možnost jednoduchého porovnání je u každého typu komprese uvedena tabulka jeho základních vlastností. Popis této tabulky ukazuje vzorová tabulka 2.1.

Ztrátovost	Uvedení, zda je komprese ztrátová. Případně co ztráty způsobují.
Vhodné	Popisuje, kdy je vhodné použít uvedený algoritmus.
Nevhodné	Popisuje, kdy naopak není vhodné algoritmus využít.
Symetrie	Symetrie mezi algoritmem komprese a dekomprese. Komprese je symetrická, pokud je doba komprese a dekomprese stejná nebo srovnatelná.
Výkon	Pokud je možné, je zde uvedena algoritmická složitost. Složitost bude uváděna v závislosti na velikosti vstupních dat n. Pro obrazovou informaci roste velikost vstupních dat kvadraticky s jejich rozlišením. Dále zde budou uvedeny případné odhady kompresního poměru.
Výhody	Výhody uvedeného kompresního algoritmu.
Nevýhody	Nevýhody uvedeného algoritmu.
Aplikace	Aplikace komprese nebo příklad formátu.

Tabulka 2.1: Vzorová tabulka vlastností kompresních algoritmů.

2.1.2 RLE – Run length endoding

Jedná se o jednoduchou kódovací metodu určenou především pro obrázky obsahující velké jednobarevné plochy. Většinou se používá u obrázků s malým barevným rozlišením a nebo v případě, kdy jsou barvy definované paletou. Podstatou kódování RLE je uložení stejných po sobě jdoucích hodnot jako číslo počtu opakování a hodnotu.

Např. pokud máme vstupní data, která mohou obsahovat pouze znaky písmen abecedy. Vstupní sekvenci

`AABBBABCAAAAABBBCCDDDDDDDDDAAAA`

o délce 30 znaků můžeme kódovat do podoby

`AA3BABC5A3BCC8D4A`.

Vstupní data jsme komprimovali na 17 znaků. Uvedenou kompresi jsme provedli tak, že skupiny stejných hodnot (běhy) o délce větší než 2 znaky[1] jsme nahradili číselnou hodnotou a znakem.

Uvedený příklad je pouze ilustrativní. Samotná implementace kódování závisí na definici vstupních dat.

[1]Kódování kratších sekvencí je zbytečné, protože nám nepřinášejí žádnou úsporu.

Ztrátovost	Bezztrátová komprese.
Vhodné	Soubory obsahující delší řady stejných hodnot. U obrázků jsou to především obrázky s malou barevnou hloubkou. Např. ručně kreslené obrázky, grafy, tabulky, černobílé obrázky apod.
Nevhodné	Data s velkým šumem a s velkým rozsahem hodnot. U obrázků jsou to např. fotky.
Symetrie	Symetrický.
Výkon	Výpočetní i paměťová složitost $O(n)$. Kompresní poměr záleží především na typu vstupních dat. Pro nevhodná data bude kompresní poměr roven jedné. V případě vhodného použití se může kompresní poměr pohybovat v řádech desítek.
Výhody	Jednoduchá implementace, rychlá komprese a dekomprese
Nevýhody	Použitelnost u malé skupiny dat.
Aplikace	Používá formát PC Paintbrush File Format (PCX). Většina složitějších kompresních algoritmů zahrnuje RLE jako svoji část.

Tabulka 2.2: Tabulka vlastností algoritmu RLE.

2.1.3 Entropie a entropické kódování

Entropické kódování je popsáno v [WE09] odkud bylo čerpáno. Entropie udává míru neurčitosti systému nebo dat. Čím je systém náhodnější, tím je entropie větší. Nás bude především zajímat entropie vstupních dat, které chceme komprimovat. Jedná se o informační entropii někdy nazývanou jako *Shannonova entropie*. Čím jsou vstupní data náhodnější, tím je entropie vstupních dat větší. Naopak pokud výskyt jednotlivých znaků má „ostré“ rozložení pravděpodobnosti nebo pokud dokonce existuje závislost mezi jednotlivými vstupními znaky, tak entropie klesá. Entropie je maximální pro rovnoměrné rozložení pravděpodobnosti výskytu jednotlivých vstupních znaků. Entropie je minimální (nulová), pokud vstupní sekvenci můžeme předem přesně určit.

Mnoho kompresních algoritmů je postaveno na myšlence, že většina vstupních dat nemá maximální entropii. Ve většině reálných dat se setkáváme s tím, že určité hodnoty se vyskytují častěji než ostatní. Např. u psaného textu převažují samohlásky jako 'A' a 'E' a znaky jako 'Q' nebo 'X' se v česky psaném textu vyskytují jen velmi zřídka. Myšlenkou těchto algoritmů je kódovat častější znaky kratším kódem a naopak znaky s malou pravděpodobností výskytu mohou mít kód delší. Takové kompresní algoritmy můžeme označit jako entropické.

Když známe přesné pravděpodobností rozložení vstupních dat, tak známe jejich informační hodnotu. Pokud navíc můžeme pro každý vstupní znak definovat kód, jehož délka je

minimální nutná pro uložení jeho informační hodnoty, tak máme optimální bezztrátovou kompresi.

2.1.4 Huffmanovo kódování

Ztrátovost	Bezztrátová komprese.
Vhodné	Pokud víme, že vstupní data obsahují hodnoty, které mají rozdílné pravděpodobnosti výskytů – některé jsou časté a některé se vyskytují jen zřídka. Rozsah vstupních hodnot by neměl být velký. Lze použít na kompresi černobílého obrazu, na skici a kreslené obrázky.
Nevhodné	Nevhodné, pokud o komprimovaných datech nemáme žádné nebo jen omezené informace. Nevhodné pro fotografie a obrazy s vysokým rozlišením.
Symetrie	Symetrický.
Výkon	Výpočetní i paměťová složitost $O(n)$. Kompresní poměr závisí především na tom, zda je pro daná data zvolena správná převodní tabulka. Kompresní poměr dosahuje hodnot 5 – 15.
Výhody	Jednoduchá implementace, rychlá komprese a dekomprese
Nevýhody	Použitelnost u malé skupiny dat. Musíme znát statistiky pravděpodobností výskytu jednotlivých vstupních znaků. Malá granularita délky kódových slov.
Aplikace	Složitější kompresní algoritmy využívají Huffmanovo kódování jako svoji součást. Lze použít v kombinaci s aritmetickým kódováním.

Tabulka 2.3: Tabulka vlastností Huffmanova kódování.

Prvním příkladem entropického kódování je Huffmanovo kódování. Je založeno na tom, že každý vstupní znak (nebo skupina znaků) je kódován jedním kódovým slovem. Kódová slova jsou tvořena tak, že čím je znak častěji používán, tím bude mít kratší kódové slovo. Protože kódová slova jsou různě dlouhá, tak musíme zařídit rozpoznání konce znaku. To je docíleno pomocí *prefixového kódování*. To znamená, že žádné kódové slovo není předponou (prefixem) jiného kódového slova.

A	0
H	101
J	100
O	11

Obrázek 2.1: Kódová tabulka.

Například vezmeme zmíněné kódování textu. Máme kódovou tabulku zobrazenou na obrázku 2.1. Je vidět, že používanější znaky jako 'A' a 'O' mají kratší kódová slova. Navíc je z příkladu patrné *prefixové kódování*, kde znak 'A' je kódován jako 0, a proto žádné další kódové slovo nesmí začínat tímto znakem.

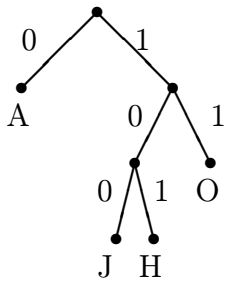
Pokud budeme chtít zakódovat sekvenci

AHOJ,

získáme kód

010111100.

Při dekódování postupně procházíme kód a jakmile rozpoznáme kódové slovo, tak ho převedeme na jeho hodnotu. Pro dekódování se většinou využívá průchod kódovým stromem. Kódový strom pro předchozí kódovou tabulku je na obrázku 2.2.



Obrázek 2.2: Kódový strom.

Při dekódování postupně procházíme stromem od kořene, dokud nenarazíme na list. Když narazíme na list, přeložíme kódové slovo, uložíme hodnotu na výstup a přesuneme se opět do kořene stromu. Pro konstrukci kódového stromu a převodní tabulky je potřeba znát pravděpodobnosti výskytu jednotlivých znaků. Tyto pravděpodobnosti se získávají ze statistik měřených pro danou skupinu dat. To znamená, že převodní tabulka a kódový strom je předem daný vždy pro určitou skupinu vstupních dat. Problémem Huffmanova kódování je, že optimální délky kódových slov nebývají celočíselné. Např. se stává, že optimální délka kódu pro 'A' je 1,2 bitu a pro 'H' to může být 2,6 bitu. Při tvorbě kódových slov tedy dochází k zaokrouhlení na celý počet bitů. Tento problém řeší aritmetické kódování (viz kapitola 2.1.5).

2.1.5 Aritmetické kódování

Ztrátovost	Bezztrátová komprese.
Vhodné	Podobně jako u Huffmanova kódování. Metoda je vhodná pro data, kde převažují určité hodnoty, ale musíme předem vědět které (nebo použít nějaké adaptivní metody).
Nevhodné	Nevhodné, pokud o komprimovaných datech nemáme žádné nebo jen omezené informace. Nevhodné pro fotografie a obrazy s vysokým rozlišením.
Symetrie	Symetrický.
Výkon	Výpočetní i paměťová složitost $O(n)$. Kompresní poměr závisí především na vhodném dělení jednotlivých podintervalů. *Aritmetické kódování se přibližuje optimální bezztrátové kompresi*, pokud známe pravděpodobnost výskytu jednotlivých vstupních znaků.
Výhody	Vysoká efektivita.
Nevýhody	Je patentována.
Aplikace	Je možné ji použít jako součást ztrátových kompresních metod.

Tabulka 2.4: Tabulka vlastností aritmetického kódování.

Aritmetické kódování je druhým příkladem entropického kódování. Na rozdíl od Huffmanova kódování se kóduje celý vstupní objem dat do jediného kódového slova. Tímto kódovým slovem je číslo v intervalu $[0, 1)$.

Při kompresi začínáme s uvedeným intervalem $[0, 1)$. Interval rozdělíme na podintervaly tak, že každý podinterval odpovídá jednomu vstupnímu znaku. Četnější vstupní znaky mají větší podintervaly než méně pravděpodobné vstupní znaky. Při kódování vybereme podinterval podle vstupního znaku a ten znovu stejným způsobem dělíme. Tímto způsobem zmenšujeme výsledný interval. Po zakódování celé zprávy máme jeden malý interval hodnot. Výsledkem komprese je libovolná hodnota z výsledného intervalu.

Komprese je dána nepravidelným rozdělením intervalů. Častější hodnoty mají větší podinterval, a proto méně omezují výsledný interval. Díky tomu, že intervaly mohou mít libovolně velkou velikost, odpadá problém se zaokrouhlením, kterým trpěl Huffmanův kód.

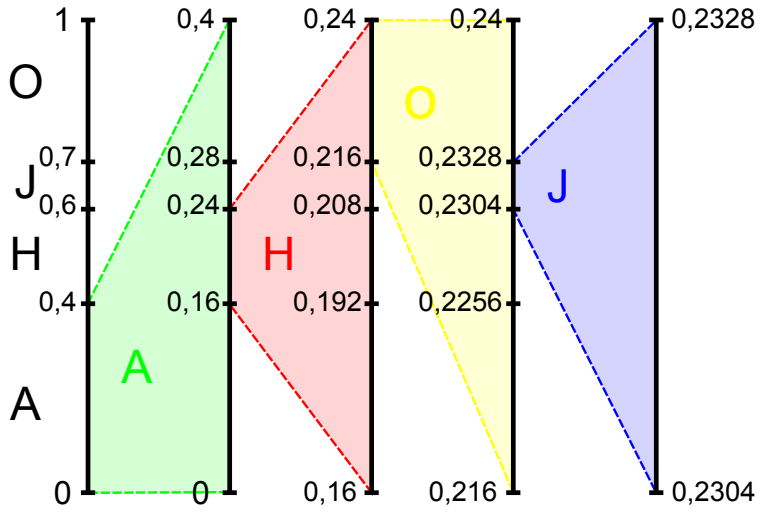
Příklad kódování je stejný jako v kapitole Huffmanova kódu (2.1.4). Tabulka 2.3 ukazuje pravděpodobnost výskytu jednotlivých vstupních znaků.

Obrázek 2.4 ukazuje příklad kódování vstupní sekvence AHOJ. Začíná se s intervalem $[0, 1)$, který se postupně zpřesňuje podle vstupní sekvence. Z obrázku je patrné stejné dělení intervalů v každém kroku. Výsledkem kódování vstupních dat je interval

A	40%
H	20%
J	10%
O	30%

Obrázek 2.3: Předpokládané rozložení znaků ve vstupním textu.

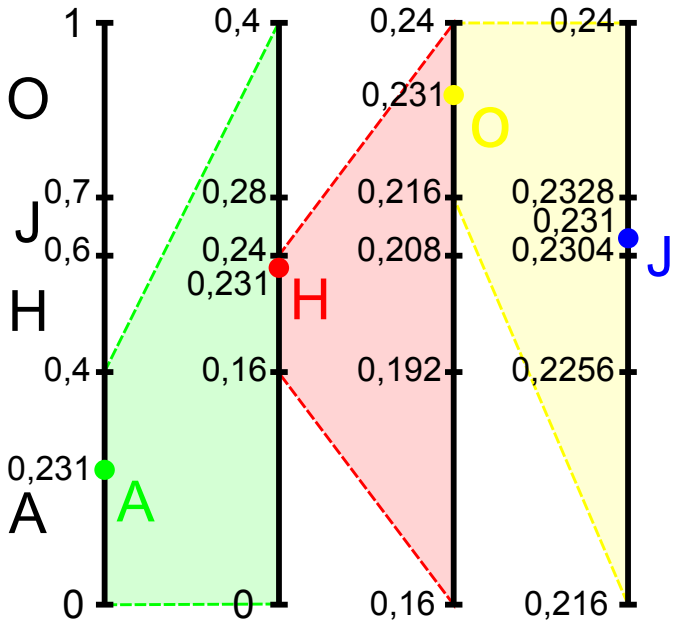
$[0,2304; 0,2328)$. Jako výsledné kódové slovo můžeme vybrat libovolnou hodnotu z výsledného intervalu. Takže výsledkem může být hodnota $0,231$ nebo $0,232$.



Obrázek 2.4: Příklad aritmetického kódování.

Dekódování z číselné hodnoty pracuje podobným způsobem. Postupně zjišťujeme, v jakém intervalu se dekódovaná hodnota nachází, a uvedený interval zpřesňujeme stejným způsobem. Na obrázku 2.5 je zobrazeno dekódování hodnoty $0,231$.

Při implementaci uvedeného algoritmu je potřeba umět pracovat s nekonečnou přesností desetinných čísel. Většinou se nedrží v paměti celá hodnota intervalu, ale pouze jeho určitá část. Jak je vidět i z uvedeného příkladu, tak se po určité době začátek čísla již nemění a mění se pouze hodnoty posledních desetinných míst. Na rozdíl od uvedeného příkladu se v reálné implementaci místo desítkové soustavy používá dvojková reprezentace čísel.



Obrázek 2.5: Příklad dekódování aritmetického kódu.

Hlavním problémem aritmetického kódování (stejně jako prakticky všech kompresních metod založených na entropii) je nutnost znalosti četnosti (pravděpodobnosti) jednotlivých vstupních znaků. Je možné mít tyto četnosti zadané vždy pro danou skupinu vstupních dat. Existují také adaptivní algoritmy, které postupně mění velikosti jednotlivých intervalů v závislosti na zpracovávaných vstupních datech.

2.1.6 Slovníková komprese

Poměrně jednoduchá bezztrátová kompresní metoda. Jedná se o algoritmus známý jako LZW[2]. Již z názvu komprese je patrné, že komprese využívá slovník. Jedná se o kódovou tabulku obsahující vzorky dat a jim odpovídající binární kódy.

Na začátku obsahuje slovník všechny vstupní znaky, kde binární hodnota reprezentuje přímo hodnotu znaku. Máme-li např. kódovat text obsahující základní sadu 26 znaků, bude kódová tabulka na počátku komprese obsahovat právě 26 záznamů. Při kódování se postupně procházejí data. Vždy se rozpozná nejdelší možné kódové slovo, které máme

[2]Zkratka je složena z počátečních písmen jmen autorů algoritmu.

Ztrátovost	Bezztrátová komprese.
Vhodné	Pokud data obsahují opakující se sekvence. Vhodné pokud mají data malý rozsah dat – obrázky s malým rozlišením. Vhodnější pro větší objemy dat.
Nevhodné	Nevhodné pro fotografie a obrazy s vysokým rozlišením a velkým šumem.
Symetrie	Symetrický.
Výkon	Výpočetní i paměťová složitost $O(n \log(n))$.
Výhody	Jednoduché na implementaci.
Nevýhody	Většina algoritmů patentovaných.
Aplikace	Používá se např. ve formátech GIF nebo TIFF.

Tabulka 2.5: Tabulka vlastností slovníkové komprese.

v tabulce. Toto slovo se zakóduje příslušnou hodnotou a do tabulky se uloží nové slovo složené ze slova, které se aktuálně kódovalo, a následujícího znaku.

Například máme zakódovat sekvenci znaků

ABDABCDA

a slovník při začátku kódování je zobrazen na obrázku 2.6.

A	1
B	2
C	3
D	4
..	..
Y	25
Z	26

Obrázek 2.6: Kódová tabulka na začátku komprese.

Podrobný průběh kódování uvedeného textu popisuje obrázek 2.7. Výstupní kódová sekvence je

1 2 4 27 3 29.

Výhodou tohoto kódování je, že není potřeba přenášet slovník s daty, protože je možné ho při dekompresi sestrojit podobným způsobem. Postupně se dekódují jednotlivá kódová slova pomocí slovníku. Do slovníku se vkládá nové slovo složené z předchozího dekódovaného slova a prvního znaku posledního dekódovaného slova. Může nastat situace, kdy

Nejdelší nalezené slovo	Kódovaný znak	Výstupní kód	Nové slovo slovníku
–	A	–	–
A	B	1	AB=27
B	D	2	BD=28
D	A	4	DA=29
A	B	–	–
AB	C	27	ABC=30
C	D	3	CD=31
D	A	–	–
DA	–	29	–

Obrázek 2.7: Ukázka slovníkové komprese.

kódové slovo není uložené ve slovníku, a jeho hodnota je o jedna větší než poslední hodnota ve slovníku. To může nastat jen v případě, že kódované slovo začíná a končí stejným znakem. V tom případně se vloží do slovníku slovo složené z posledního dekódovaného slova a jeho prvního písmena přidaného na konec slova. Ukázka dekódování předchozího příkladu je uvedena na obrázku 2.8.

Vstupní kód	Předcházející slovo	Dekódované slovo	Nové slovo slovníku
1	–	A	–
2	A	B	AB=27
4	B	D	BD=28
27	D	AB	DA=29
3	AB	C	ABC=30
29	C	DA	CD=31

Obrázek 2.8: Ukázka slovníkové dekomprese.

2.1.7 Fourierova transformace

Fourierova transformace je transformace signálu do duálního prostoru. Vstupní signál se vyjádří jako součet harmonických signálů. Tedy jako součet funkcí sinus a cosinus. Obecně se jedná o funkce komplexní exponenciály. Při kompresi dat se využívá tzv. diskrétní Fourierova transformace a speciálně diskrétní kosinová transformace. Což je jedna z forem Fourierovy transformace. Podrobný popis uvedených transformací je možné nalézt v [ŽBSF04, WF10].

Ztrátovost	Možnost ztrátové i bezztrátové komprese.
Vhodné	Obrazová data s větším barevným rozlišením. Vhodné zejména pro fotografie.
Nevhodné	Obrazy s malým barevným rozlišením. Nevhodné pro ruční kresby, skenovaný text a podobně. Problémem bývají ostré hrany v obrazech.
Symetrie	Symetrický.
Výkon	Výpočetní složitost rychlé Fourierovy transformace je $O(n \log (n))$. Paměťová složitost $O(n)$. Dosahuje kompresního poměru 20:1 až 25:1.
Výhody	Umožňuje dosáhnout většího kompresního poměru.
Nevýhody	Závisí většinou na aplikaci. U formátu JPEG je nevýhodou dělení obrazu na makrobloky.
Aplikace	Používá se především ve formátu JPEG.

Tabulka 2.6: Tabulka vlastností komprese za pomoci Fourierovy transformace.

Vstupem diskrétní transformace je konečný počet vzorků dat. Jsou to vstupní hodnoty, které chceme komprimovat. Výstupem jsou potom parametry jednotlivých harmonických signálů. V případě diskrétní kosinové transformace se jedná o amplitudy kosinových signálů, které mají různou frekvenci. Komprese je potom založená na tom, že pro pozorovatele jsou nejdůležitější nízké frekvence. První parametr, takzvaná stejnosměrná složka, udává průměrnou hodnotu vstupních hodnot. Následují parametry od nízkých po vyšší frekvence. Vysoké frekvence mají většinou malé hodnoty a mohou být zaokrouhleny nebo úplně zanedbány, aniž by došlo k velké změně nebo ztrátě dat. Použití transformace bude ukázáno na formátu JPEG, který tuto transformaci využívá.

Formát JPEG převádí obrazová data do barevného modelu YC_BC_R. Barevné složky C jsou podvzorkovány. Obrazová data jsou rozdělena do čtverců 8×8. Jednotlivé složky každého čtverce jsou transformovány pomocí diskrétní kosinové transformace. Výsledkem kosinové transformace je sada parametrů kosinových funkcí. Jedná se o reálná čísla a počet parametrů je stejný jako počet vstupních hodnot. Jedná se opět o matici 8×8. Z těchto hodnot je možné sestrojit původní data bez ztráty informace. Ke ztrátě určité části informace dochází v kroku kvantizace těchto parametrů. Kvantizací se zde rozumí vydělení matice parametrů kvantizační maticí, která je pro formát JPEG přesně definována. Uživatel navíc může zvolit parametr kvality. Kvantovaná data jsou poté kódována. Pro kódování se používá většinou Huffmanovo nebo aritmetické kódování.

Celá myšlenka je založena na tom, že výsledkem kvantizace je velmi řídká matice s dominantním levým rohem. Při kvantizaci dochází ke zmíněnému zanedbání vysokých frekvencí. Řídká matice se potom velmi dobře komprimuje pomocí bezztrátové komprese.

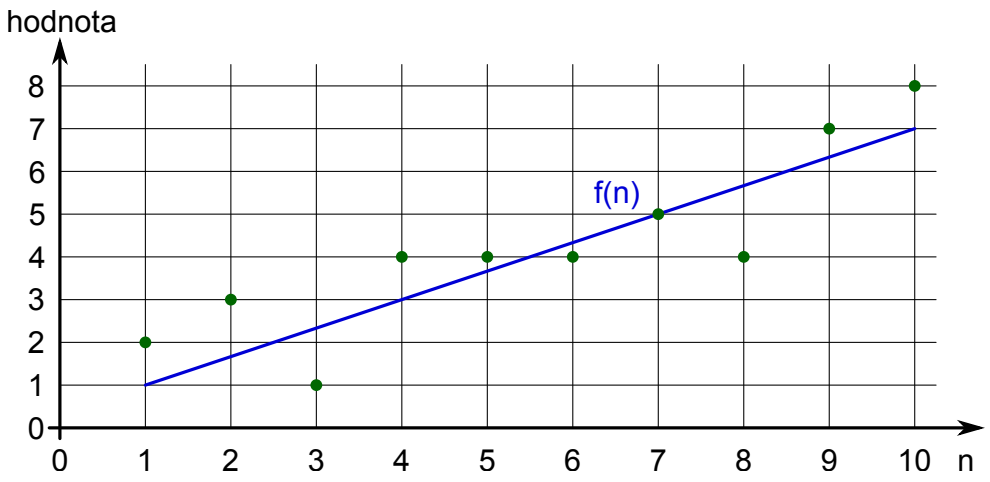
Asi největším problémem JPEG komprese je dělení dat na bloky. Tyto problémy odstraňují přístupy transformace celých dat najednou. Fourierova transformace je potom nahrazena transformací za pomoci tzv. vlnek (wavelet).

2.1.8 Zjednodušení dat

Další myšlenka, velmi často využívaná v kompresi dat, je možnost zjednodušení dat. Zjednodušením se zde rozumí především zmenšení rozsahu vstupních hodnot. Pokud se nám podaří zmenšit rozsah hodnot, můžeme je kódovat za pomoci kratších kódů a tím dosáhneme komprese dat. Zjednodušení se provádí na základě podobnosti. Předpokládáme, že komprimovaná data jsou podobná jiným hodnotám, které jsou nám známé.

Podobnost matematické funkci

Prvním příkladem může být podobnost matematické funkci. Máme data a víme, že jejich hodnoty se podobají průběhu lineární funkce. Na obrázku 2.9 je zobrazeno devět vstupních hodnot (zelené body) proložených lineární funkcí $f(n)$ (modrá úsečka).



Obrázek 2.9: Příklad podobnosti matematické funkci.

Pro rovnici lineární funkce je v našem příkladu platí:

$$f(n) = \frac{2}{3}y + \frac{1}{3}$$

Vstupní hodnoty jsou pouze celá čísla. Stejně tak hodnotu funkce vždy zaokrouhlíme na celé číslo. Na obrázku 2.10 jsou zobrazeny vstupní hodnoty, hodnoty funkce a rozdíl uvedených hodnot.

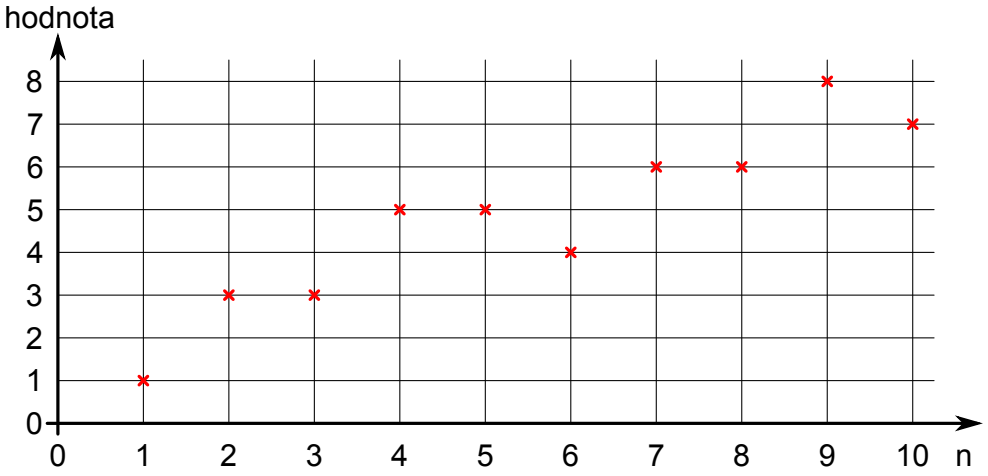
n	1	2	3	4	5	6	7	8	9	10
hodnota	2	3	1	4	4	4	5	4	7	8
$f(n)$	1	2	2	3	4	4	5	6	6	7
rozdíl	1	1	-1	1	0	0	0	-2	1	1

Obrázek 2.10: Porovnání vstupních hodnot a matematické funkce.

Rozsah vstupních hodnot je 1–8, zatímco rozsah rozdílu hodnot a naší funkce je -2–1. Rozsah hodnot rozdílu je poloviční. K dekódování potom potřebujeme hodnoty rozdílů a parametry matematické funkce. V našem příkladu jsme použili lineární funkci a tu lze popsat dvěma parametry 2/3 a 1/3.

Podobnost sousedních hodnot

Druhým příkladem podobnosti je podobnost hodnoty se svým okolím. Jinými slovy, že rozdíl sousedních hodnot je malý. Velmi často mají vstupní data velký rozsah hodnot, ale změna mezi sousedními daty je velmi malá. Na obrázku 2.11 jsou vstupní hodnoty.



Obrázek 2.11: Příklad podobnosti sousedních hodnot.

Obrázek 2.12 zobrazuje vstupní hodnoty a rozdíl s předcházející hodnotou. První hodnota je použita přímo.

n	1	2	3	4	5	6	7	8	9	10
hodnota	1	3	3	5	5	4	6	6	8	7
rozdíl	1	2	0	2	0	-1	2	0	2	-1

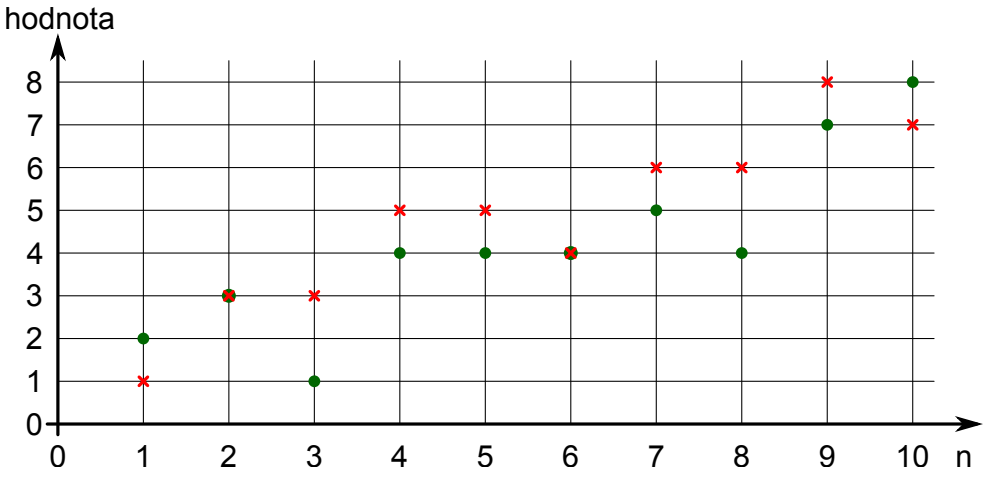
Obrázek 2.12: Porovnání vstupních hodnot a jejich rozdílů.

Výsledek je v tomto příkladu shodný s předchozím příkladem. Rozsah vstupních hodnot je 1–8, ale změna hodnoty oproti předchozí je v rozmezí -1–2. Rozsah hodnot se opět zmenšil na polovinu. Výhodou tohoto zjednodušení je, že pro zpětnou rekonstrukci dat nepotřebujeme žádné další informace, jako byly parametry rovnice v minulém příkladu.

Pokud však data budou obsahovat velké změny hodnot, tak místo k zjednodušení a zmenšení rozsahu dat dojde k zvětšení datového rozsahu. Pokud by vstupní data obsahovala sekvenci 8–1–8, tak by rozdíly sousedních hodnot byly -7–7. Takže rozsah hodnot vzrostl na 15.

Podobnost mezi různými daty

Další možností je, že různá vstupní data jsou si vzájemně podobná. Obrázek 2.13 ukazuje porovnání vstupních hodnot z předchozích dvou příkladů. Byly zachovány značky i barvy hodnot, takže hodnoty z prvního příkladu jsou zobrazeny jako zelená kolečka a hodnoty z druhého jako červené křížky.



Obrázek 2.13: Příklad podobnosti různých dat.

Na obrázku 2.14 je srovnání vstupních dat. Tabulka obsahuje hodnoty obou vstupů a jejich rozdíl.

n	1	2	3	4	5	6	7	8	9	10
1. data	2	3	1	4	4	4	5	4	7	8
2. data	1	3	3	5	5	4	6	6	8	7
rozdíl	1	0	-2	-1	-1	0	-1	-2	-1	1

Obrázek 2.14: Porovnání dvou různých dat.

Stejně jako v obou předchozích příkladech má rozdíl hodnot poloviční rozsah hodnot.

Při výpočtu rozdílu dvou různých dat je většinou vhodné provést nějakou transformaci celých dat nebo jejich části. Např. při kompresi videa nebo animace lze předpokládat, že sousední snímky jsou si podobné. Při pohybu kamery se liší zejména posunutím. Pokud budeme schopni určit vektor posunutí a provedeme výpočet rozdílu dvou snímků s tím, že první posuneme o vektor pohybu, tak získáme mnohem lepší výsledky.

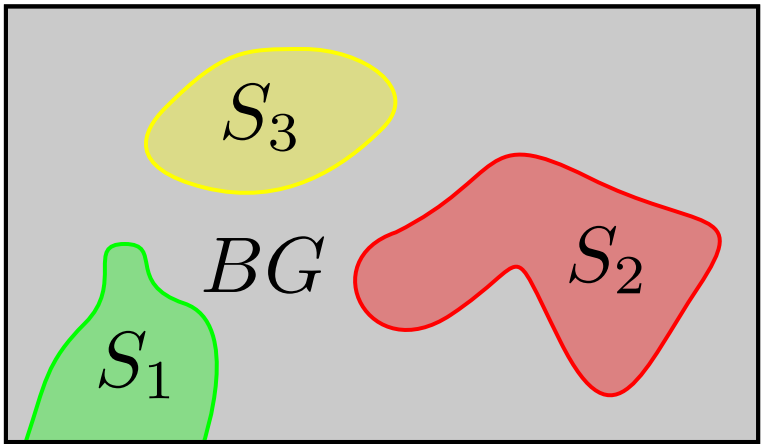
2.2 Segmentace a dekompozice obrazu

V této kapitole bylo čerpáno především z [ŠHB08].

2.2.1 Úvod do segmentace

Segmentací rozumíme rozčlenění obrazového snímku na oblasti (segmenty) S_i, které mají souvislost s předměty reálného světa. Jednotlivé segmenty by měly být disjunktní oblasti. Jinými slovy, žádné dva segmenty nesmí obsahovat stejný bod obrazu. Pokud jednotlivé segmenty jednoznačně korespondují s objekty na vstupním obrazu, hovoříme o úplné nebo kompletní segmentaci. V opačném případě se jedná o částečnou segmentaci. Úplná segmentace je podstatná například v oblastech umělé inteligence a rozpoznávání obrazu. Nás bude zajímat spíše segmentace částečná, která dělí obraz na části, které mají určité společné vlastnosti, jako jsou jas, barva nebo struktura.

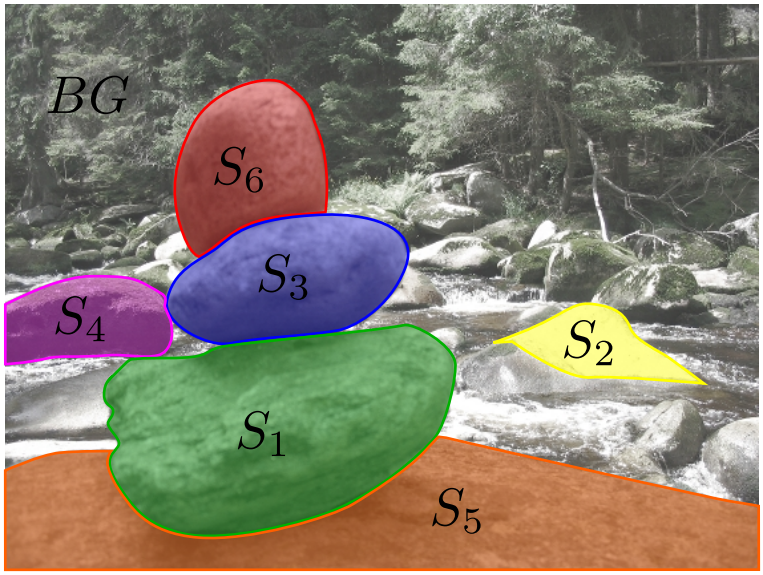
Na obrázku 2.15 je schématicky znázorněná segmentace obrazu. Všechny body obrazu označíme jako IM (obraz – image). S_i je označení jednotlivých segmentů. Pozadí (background) je označeno jako BG.



Obrázek 2.15: Schématické znázornění segmentace obrazu.

Obrázek 2.16: Výchozí obrázek používaný v příkladech.

Příklady segmentace budou vysvětlovány na obrázku 2.16. Pro úpravu obrázků v příkladech byla používána zejména aplikace [GIMP]. Na následujícím obrázku 2.17 je znázorněno, jaký výsledek očekáváme od segmentace tohoto obrazu.



Obrázek 2.17: Příklad segmentace obrazu.

Uvedené informace můžeme formulovat pomocí následujících rovnic:

$$IM = \left(\bigcup_{i=1}^{n} S_i \right) \bigcup BG$$

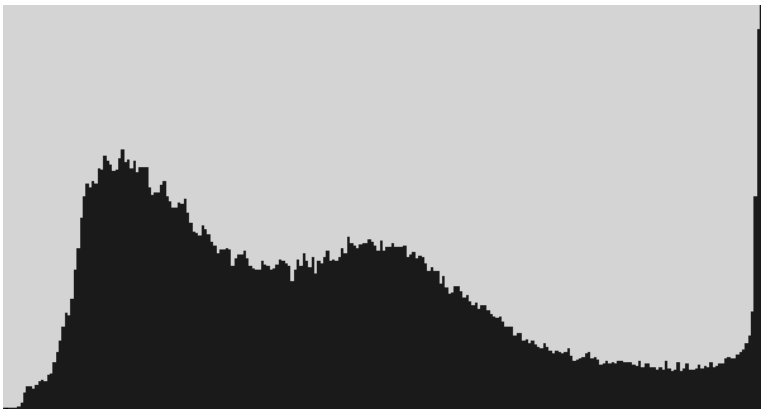
$$S_k \bigcap S_l = \{\emptyset\} \quad \text{pro} \quad \forall k \neq l$$

Metody segmentace lze rozdělit do čtyř hlavních skupin:

1. Globální metody, které většinou využívají histogram segmentovaného snímku.
2. Metody založené na hledání hran jednotlivých segmentů.
3. Metody, které přímo vytvářejí oblasti podle podobných vlastností sousedních bodů.
4. Kombinace předchozích metod. Jedná se o metody, které se pokoušejí použít jak informace o hranách, tak porovnávají i vlastnosti jednotlivých oblastí.

2.2.2 Histogram obrazu

Histogram je globální charakteristika snímku. Zobrazuje se jako graf četností jasů nebo barev ve snímku. Histogram ukázkového obrázku 2.15 je zobrazen na obrázku 2.18.



Obrázek 2.18: Histogram obrazu.

Pokud jas bodu o souřadnicích x, y označíme jako $i(x, y)$, můžeme definovat funkci pro porovnání jasu jako:

$$P_n[i(x,y)] = \begin{cases} 1 & \text{pro} \quad i(x,y) = n \\ 0 & \text{pro} \quad i(x,y) \neq n \end{cases}$$

Potom lze histogram definovat absolutně jako funkci H:

$$H(n) = \sum_{\forall(x,y)\in IM} P_n[i(x,y)]$$

Při této definici platí rovnice:

$$\sum_{n=min}^{max} H(n) = \text{plocha snímku v obrazových bodech}$$

Relativní definice histogramu je:

$$H_r(n) = \frac{1}{\sum\limits_{n=min}^{max} H(n)} \cdot \sum_{\forall(x,y)\in IM} P_n[i(x,y)]$$

Pro relativní definici platí podobný vztah jako pro absolutní rovnici:

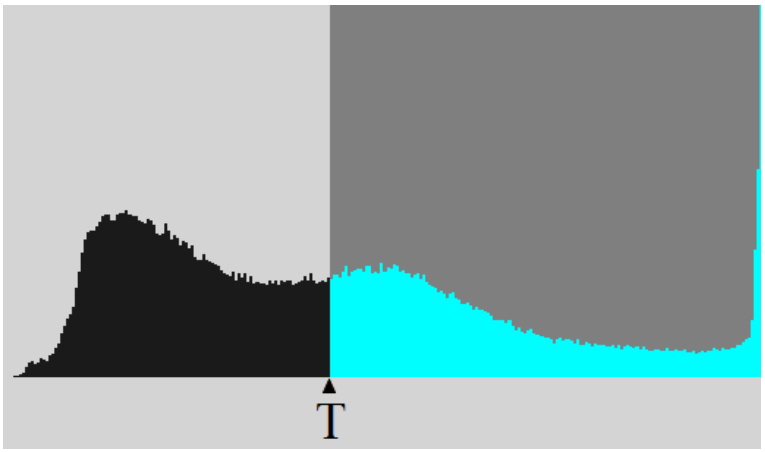
$$\sum_{n=min}^{max} H_r(n) = 1$$

$H_r(n)$ potom udává pravděpodobnost, že náhodně vybraný bod bude mít jas n.

2.2.3 Segmentace prahováním

Nejjednodušší metody segmentace vycházejí z histogramu obrazu. Jedná se o globální metodu. Body se přiřazují do oblasti podle jasu. Základem je prahování s jedním prahem.

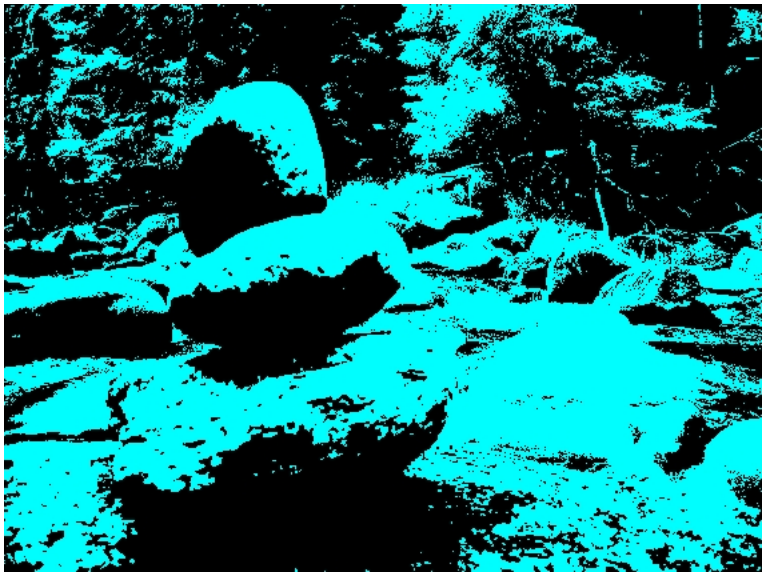
Nejprve se určí prahová hodnota T. V některých případech se práh určuje ručně, ale existují metody pro automatické určení prahu. Histogramy bývají velmi často bimodální. To znamená, že na histogramu jsou dvě lokální maxima. V okolí jednoho se nacházejí body, které přísluší pozadí obrázku. V okolí druhého potom můžeme předpokládat body v popředí. Prahová hodnota se potom volí mezi těmito maximy, např. do lokálního minima mezi nimi. Je možné histogram upravit pro jednodušší automatické hledání prahu například tím, že z histogramu vyřadíme body, které leží na hranicích objektů. Jedná se o body s velkou hodnotou gradientu. Touto úpravou histogramu lze dosáhnout většího rozdílu mezi maximy a minimem. Literatura popisuje mnoho dalších možných úprav histogramu pro zjednodušení a zlepšení automatického vyhledávání optimálního prahu.



Obrázek 2.19: Volba prahu v histogramu.

Příklad volby prahu v histogramu je zobrazen na obrázku 2.19. Výsledek prahování zobrazuje obrázek 2.20, kde je vybraný segment zabarven světle modrou barvou a pozadí má barvu černou. Pokud bychom označili body segmentu hodnotou 1 a barvu pozadí hodnotou 0, tak můžeme zapsat funkci prahování $g(x, y)$ následujícím způsobem:

$$g(x,y) = \begin{cases} 1 & \text{pro} \quad i(x,y) \geq T \\ 0 & \text{pro} \quad i(x,y) < n \end{cases}$$



Obrázek 2.20: Prahování jedním prahem.

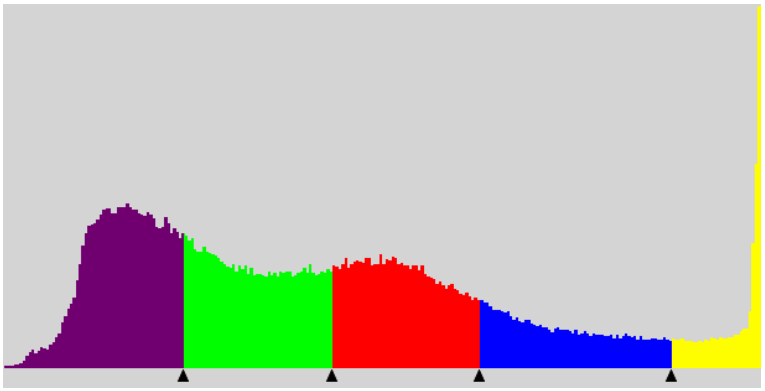
Podobným způsobem lze provádět prahování za pomoci více prahů. Funkce prahování pro více prahů:

$$g(x,y) = \begin{cases} 1 & \text{pro} \quad i(x,y) \leq T_1 & i(x,y) \in S_1 \\ 2 & \text{pro} \quad i(x,y) \in (T_1, T_2) & i(x,y) \in S_2 \\ 3 & \text{pro} \quad i(x,y) \in (T_2, T_3) & i(x,y) \in S_3 \\ & \vdots & \vdots \\ k & \text{pro} \quad i(x,y) \in (T_{k-1}, T_k) & i(x,y) \in S_k \end{cases}$$

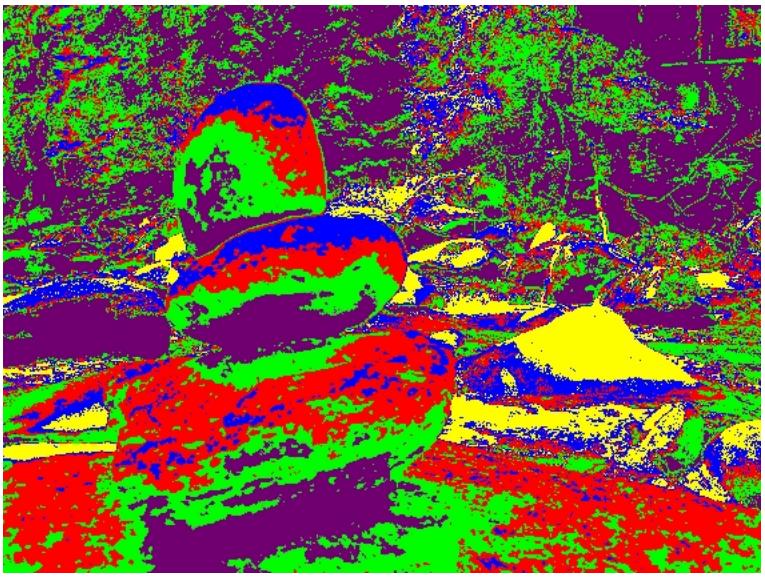
Volba prahů se provádí opět ručním výběrem nebo automaticky. Automatická volba prahů je v tomto případě komplikovanější. Ukázka volby několika prahů zobrazených v histogramu je na obrázku 2.21. Obrázek po prahování je na obrázku 2.22. V příkladu byly použity čtyři prahy. Výsledný obrázek je složen z pěti segmentů. Každý segment je označen vlastní barvou, která odpovídá barvě v histogramu.

V přechozích příkladech bylo provedeno prahování na základě jasové informace bodu. Vstupní data jsou většinou barevné obrazy, které jsou uloženy jako tři barevné složky. Prahování může využívat histogramy všech barevných složek. Ukázkou je obrázek 2.23, kde byl původní obrázek 2.15 rozdělen na složky YC_BC_R a prahována pouze složka C_R.

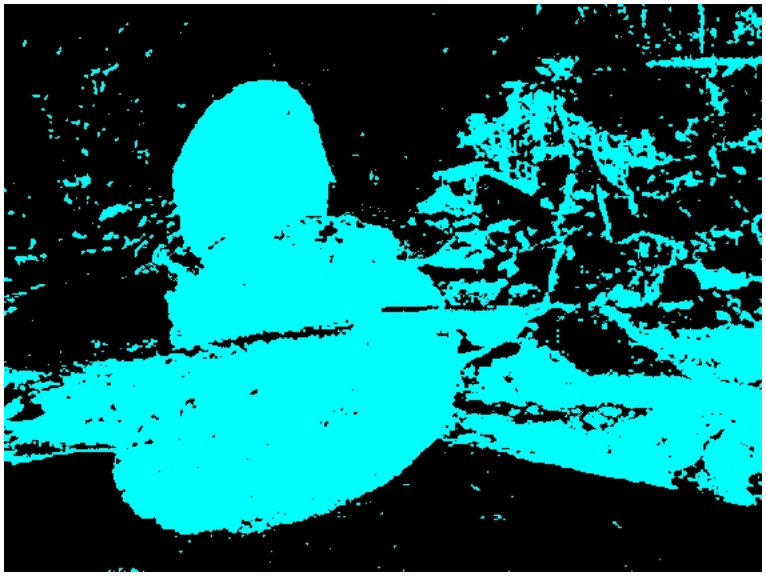
Výhodou prahování je jeho jednoduchost a rychlost výpočtu. Nevýhodou je velký vliv šumu. Výsledky segmentace prahováním obecně nedosahují dobrých výsledků. Prahování lze použít u obrazů, které obsahují kontrastní objekty a nekontrastní pozadí.



Obrázek 2.21: Volba více prahů v histogramu.



Obrázek 2.22: Prahování více prahy.



Obrázek 2.23: Prahování složky C_R.

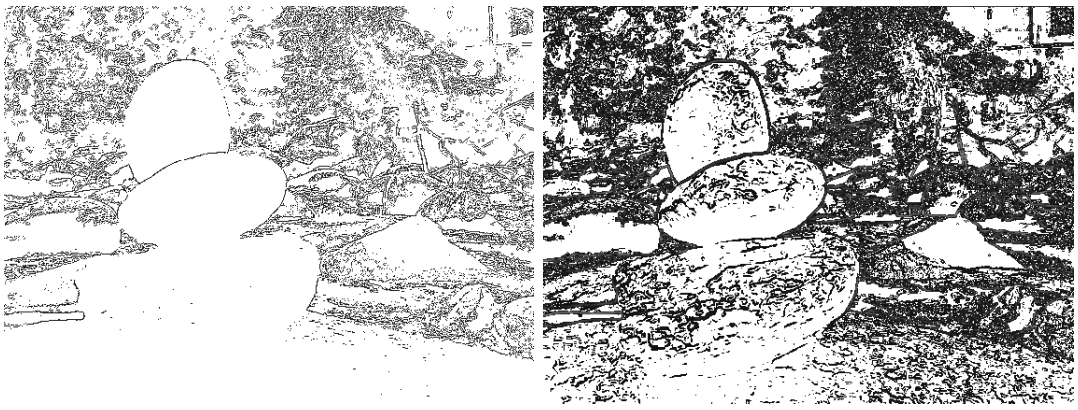
2.2.4 Hledání hran v obrazu

Hranou v obrazu rozumíme bod, ve kterém se mění jas vůči okolním bodům. Směr růstu jasu v bodě x, y se označuje jako gradient – $\nabla i(x, y)$ nebo grad $i(x, y)$. Směr hrany je potom kolmý na směr gradientu. Velikost změny v určitém směru je označován jako derivace ve směru. Největší hodnota derivace je ve směru gradientu. U diskrétní funkce potom derivaci přibližně nahrazujeme diferencí – rozdílem hodnot.

Metody hledání hran jsou většinou lokální. To znamená, že pro každý bod testují pouze jeho blízké okolí. Stejně jako většinu ostatních filtrů obrazu lze i hledání hran zobecnit jako diskrétní konvoluci. Typ filtrace je potom popsán pomocí takzvaného konvolučního jádra. V případě diskrétní konvoluce má toto jádro prakticky vždy omezenou velikost a nazývá se maska filtrace. Masky jsou většinou čtvercové a pro filtrování obrazů mají velikosti od 2×2 do 5×5, ale mohou být i mnohem větší.

Základní metody detekce hran definují množinu masek, pomocí kterých se poté vypočítá hranovost bodu. Jsou označovány jako hranové operátory (gradient operator). Některé metody počítají pouze změnu vůči okolí a jsou nezávislé na směru hrany. Je jím například Laplaceův operátor. Jiné metody počítají hodnoty derivací (resp. diferencí) v různých směrech. Pro každý směr je definována jedna konvoluční maska. Výsledná hodnota hranovosti je poté dána jako součet nebo jako maximum z vypočtených hodnot. Mezi nejznámější operátory patří Robertsův a Sobelův operátor. Podrobný popis všech

uvedených operátorů je možné nalézt v [ŠHB08]. Nevýhodou těchto hranových operátorů je, že zkoumají poměrně malé okolí bodu a jsou velmi náchylné na šum ve vstupních datech. Ukázka použití hranových operátorů je na obrázku 2.24.



(a) Laplaceův operátor (b) Sobelův operátor

Obrázek 2.24: Použití hranových operátorů.

Hledání hran v obrazu znamená hledání lokálních extrémů (minim a maxim) první derivace jasové funkce. Extrém první derivace lze určit podle toho, že druhá derivace je nulová. Na této myšlence je založena další skupina metod pro nalezení hran.

Hledání druhé derivace je ještě více citlivější na šum než hledání diferencí v případě použití hranových operátorů. Proto je nutné nejprve obraz upravit vyhlazovacím filtrem. Vhodným filtrem pro tento účel je filtr, jehož koeficienty v konvoluční masce odpovídají 2D gaussovskému rozložení. Parametrem gaussovského rozložení je směrodatná odchylka, která udává velikost okolí, které filtr uvažuje (prakticky se určuje velikost konvoluční masky).

Po vyhlazení můžeme pro odhad druhé derivace použít Laplaceův operátor. Tento postup se nazývá LoG (Laplacian of Gaussian). Pro následné vyhledání průchodů nulou není vhodné přímo prahovat hodnoty s hodnotou blízkou nule. Vhodnější je např. použití masky 2×2, kde se testuje změna znaménka.

Druhou používanou metodou je DoG (difference of Gaussian), která aproximuje druhou diferenci za pomoci rozdílu dvou různě vyhlazených obrazů. Každý z obrazů je vyhlazen s jiným parametrem směrodatné odchylky a spočítá se rozdíl těchto dvou obrazů.

Problém všech uvedených přístupů je, že nepracují s měřítkem obrazu. Přitom je nutné, aby velikost konvoluční masky odpovídala velikosti detailů obrázku. Existují metody, které tento problém řeší, ale překračují rámec této práce.

Jedním z nejpoužívanějším filtrů je Cannyho detektor hran. Tento detektor je optimální vzhledem ke třem kritériím:

1. Dobrá detekce – detektor by měl najít tolik skutečných hran, kolik je v obrazu možné nalézt.
2. Dobré umístění – nalezená hrana by měla být co nejblíže skutečné hraně.
3. Minimální odezva – každá hrana by měla být označena pouze jednou. Zašumění obrazu by nemělo způsobovat chybně nalezené hrany.

Algoritmus Cannyho detektoru hran se skládá ze čtyř kroků:

1. Redukce šumu – vyhlazení za pomoci gaussovského filtru.
2. Nalezení velikosti a směru gradientu. To je možné za pomoci některého z hranových operátorů.
3. Zúžení hran – filtrace ne-maximálních hodnot ve směrech kolmých na hranu (směr gradientu a opačný směr)
4. Prahování – jedná se o takzvané prahování s hysterezí. Toto prahování zvýhodňuje body, které leží na hraně a sousedí s některým bodem s větším gradientem. Naopak body, které mají velký gradient, ale všechny sousední body mají gradient malý, jsou považovány za šum, a budou znevýhodněny.

Použití Cannyho detektoru hran na příkladu je na obrázku 2.25[3].

Uvedené detektory hran hledají hrany pouze podle hodnoty jasu. Jako vstup používají šedotónový obraz. Existují metody pro hledání hran v barevných obrazech. Další možnosti hledání hran a podrobnější popis uvedených metod je možné nalézt například v [ŠHB08, Umb05].

[3]Pro vytvoření uvedeného obrázku bylo použito externího programu [Canny].

Obrázek 2.25: Použití Cannyho detektoru hran.

2.2.5 Hledání hran segmentů

V předchozí části bylo popsáno, jakým způsobem je možné nalézt hrany obrazu. Tyto hrany nemohou být použity přímo jako výsledky segmentace, protože velmi často obsahují nežádoucí hrany, a naopak některé podstatné hrany segmentů nejsou nalezeny. Je to způsobeno většinou zašuměním vstupního obrazu nebo nedostačující informační hodnotou vstupních dat (např. pokud hledaný objekt splývá s pozadím, to může být způsobeno špatným osvětlením snímku). Tato část obsahuje popis metod pro hledání segmentů z nalezených hran.

Prahování hran

Prvním krokem úpravy nalezených hran může být jejich prahování. Obyčejné prahování nedosahuje příliš dobrých výsledků. Proto se využívá upravených metod. Jedná se zejména o dva algoritmy používané v Cannyho detektoru hran.

Prvním je zúžení hran. To se provádí tak, že pro každý bod hrany zjistíme směr procházející hrany (většinou z gradientu) a porovnáme jeho hranovost se sousedními body ve směru kolmém na směr hrany. Pokud některý ze dvou sousedních bodů má větší hranovost, bude hranovost zkoumaného bodu nulována (označení, že se nejedná o hranu).

Druhou metodou je prahování s hysterezí. Při tomto prahování jsou nastaveny dva prahy. Pokud hranovost bodu leží pod menším prahem, je jeho hranovost nulována. Při hranovosti větší než horní práh, je bod automaticky přijat jako hrana. V rozmezí mezi prahy se rozhodujeme podle sousedních bodů. Pokud v okolí bodu leží další body označené jako hrany, je bod považován za hranu, v opačném je hranovost bodu nulována. Uvedený postup je možné několikanásobně opakovat tak dlouho, dokud se dochází ke změně nalezených hran.

Věrohodnost hran

Další metodou pro opravu hran je metoda věrohodnosti hrany. Hlavní myšlenka staví na tom, že pokud má bod malou hranovost, ale leží mezi konci dvou hran, tak je velmi věrohodné, že se jedná o součást této hrany. Naopak pokud máme bod s velkou hranovostí, ale v jeho okolí se nevyskytuje žádná další hrana, bude se jednat pravděpodobně o šum.

Uvedená metoda testuje nalezené hrany. Zkoumá zakončení jednotlivých hran a jejich blízké okolí. Opakovaně se všechny hrany testují. Podle kontextu zakončení hrany vůči okolí se hraně zvětšuje nebo zmenšuje věrohodnost. Např. pokud je hrana volná bez jakéhokoliv kontextu, tak je považována za nevěrohodnou. Naopak pokud je zjištěno, že hrana navazuje na jiné hrany, tak se její věrohodnost zvětšuje. Většinou se zavádí prahové hodnoty věrohodnosti, kdy je hrana zamítnuta nebo potvrzena jako věrohodná.

Sledování hranice

Pokud známe jednotlivé oblasti obrazu (získané například segmentací), ale neznáme jejich hranice, můžeme je jednoznačně detekovat. Vstupem pro následující algoritmus je obraz s binárními hodnotami nebo máme jinak označené oblasti (např. každá část má jinou hodnotu). Hranice může být vnitřní nebo vnější. Vnitřní hranice je součástí regionu, vnější naopak není.

Algoritmus nejprve nalezne jeden bod hledané hranice (např. levý horní roh) postupným průchodem obrazem. Poté se postupuje v jednom směru – po nebo proti směru hodinových ručiček – kolem hranice oblasti. Takto prohledáváme hranici tak dlouho dokud nenarazíme znovu na výchozí bod prohledávání. Tímto způsobem je možné nalézt vnitřní i vnější hranici hledaného segmentu.

Nevýhodou této metody je, že nenajde díry uvnitř prohledávaného segmentu, ale pouze jeho vnější hranici. Hranici díry (nebo více děr) je možné nalézt stejným způsobem jako hranici oblasti.

Hledání v grafu

Známe-li předem body, kde začíná nebo končí hrana, ale neznáme její průběh, můžeme ji hledat za pomoci algoritmu hledání nejkratší cesty. Nejprve je nutné ohodnotit jednotlivé body obrazu. Správné ohodnocení je základem k správné funkci této metody. Nejjednodušší ohodnocením je obrácená hodnota hranovosti bodu, kdy hrana má potom nejmenší hodnotu. Když máme výchozí bod, koncový bod a ohodnocení jednotlivých bodů, můžeme aplikovat algoritmus hledání nejkratší cesty v ohodnoceném grafu. Pro zlepšení výkonu hledání je možné využívat heuristických metod.

Vytváření segmentů

Pokud se nám podaří nalézt uzavřené hranice objektů, je nalezení segmentů jednoduchou záležitostí. Problém nastává v případě, kdy se nám podaří získat pouze částečné hranice. Existují metody pro získání segmentů z hranic, které nejsou uzavřené. Tyto metody vycházejí z toho, že segmenty leží mezi částečnými hranicemi. Pokud tedy nalezneme dvě sousední hrany, bude segment pravděpodobně ležet mezi nimi. Tyto metody poskytují dobré výsledky jen v určitých případech a obecně se na ně nemůžeme spoléhat.

2.2.6 Segmentace zvětšováním oblastí

Předchozí text popisoval hledání hranic oblastí. Nyní se zaměříme na přímé hledání segmentů. Tyto dvě úlohy jsou duální. Je jednoduché vytvořit jednotlivé oblasti z jejich hranic a naopak segmentům přiřadit jejich hranice. Každý přístup však poskytuje jiné výsledky a každý je vhodný pro různá vstupní data. Přímá konstrukce oblastí je vhodná v případě více zašuměných dat nebo když je těžké rozpoznat hranice objektů.

Oblasti se hledají podle stejných nebo podobných parametrů jednotlivých bodů. Jinými slovy hledáme body, které jsou homogenní. Oblasti můžeme vytvářet s ohledem na různé parametry jako například jas (resp. úroveň šedi), barva, struktura, tvar atd. Na vytvářené segmenty jsou kladeny dva protichůdné požadavky:

1. Segment by měl být co nejvíce homogenní. To znamená, že rozdíl parametrů dvou libovolných bodů jednoho segmentu by měl být minimální.
2. Segment by měl být co největší.

Při vytváření segmentů je nejdůležitější vhodná volba parametrů homogenity. V našem případě budeme volit jasovou složku bodu.

Spojování oblastí

Nejjednodušším možným způsobem segmentace je porovnávání jednotlivých oblastí a jejich spojování. Nejprve musíme obraz rozdělit na jednotlivé nejmenší segmenty. Nejmenší segment může být jeden pixel nebo můžeme začít s oblastmi 2×2 nebo většími. Potom postupně procházíme všechny segmenty a porovnáme je s jejich sousedy. Pokud mají podobné nebo stejné parametry (např. průměrný jas), tak tyto dva segmenty spojíme do jednoho. Tento postup opakujeme tak dlouho dokud dochází ke spojování jednotlivých segmentů. Důležité je zmínit, že záleží na pořadí v jakém porovnáváme a spojujeme jednotlivé segmenty.

Při spojování jednotlivých segmentů můžeme přihlížet k dalším parametrům, které nám mohou pomoci při rozhodování, které segmenty budeme slučovat. Jedná se například o délku vnitřní hranice, délku hrany, se kterou segmenty sousedí, nebo přímé porovnání sousedních bodů jednotlivých segmentů.

Dělení oblastí

Jedná se o opačný přístup vzhledem k předchozímu. Segmentaci začínáme s jedním segmentem, a tím je celý obraz. V každém kroku se pokoušíme rozdělit segment na menší části. Nejprve musíme zjistit, zda dělený segment není již homogenní. Pokud není homogenní, rozdělíme ho na části a pokračujeme rekurzivně. Zde musíme velmi dobře zvážit kritéria homogenity.

Zvolíme-li jako kritérium, že dělíme segment jen pokud jednotlivé části mají různý odstín šedi, tak narazíme na problém v případě černobílých pruhů nebo šachovnice, kdy všechny části mají stejnou průměrnou hodnotu šedé, a proto se obraz nebude segmentovat. Podobný problém by byl u všech obrazů, které obsahují v jednotlivých částech vyvážený průměr šedé barvy. Lepším kritériem by bylo v tomto případě dělení segmentu v závislosti na rozptylu hodnot šedé v daném segmentu.

Dělení a spojování

Jedná se o spojení dvou předchozích metod, abychom mohli využít výhodou obou. Pro reprezentaci dělení a spojování se většinou využívá tzv. „Quadtree“. Jedná se o datovou strukturu reprezentující strom, kde každý vrchol má buďto právě čtyři potomky nebo je listem. Pokud mají všichni čtyři potomci vrcholu stejné parametry a jsou homogenní, jsou spojeni do vrcholu. Pokud naopak vrchol není homogenní, je rozdělen vytvořením potomků.

Použití této metody spočívá právě v dělení a spojování jednotlivých částí reprezentovaných pomocí stromu. Jakmile výsledný strom nemůžeme dělit ani spojovat, můžeme spojit listy, které splňují podmínku homogenity do jednoho segmentu. V tomto kroku můžeme požadovat spojité nebo nespojité oblasti, záleží na požadavcích segmentace.

Dobrým vylepšením této metody je použití dělení na segmenty, které nejsou disjunktní. Reprezentace je opět pomocí stromu, kde každý potomek má více potenciálních předků. Při konstrukci stromu potom postupujeme od nižších vrstev a vybíráme nejvhodnějšího předka ze všech možných. Existuje více algoritmů a možností této metody.

Segmentace rozvodím

Obraz si můžeme představit jako funkci ve dvou rozměrech. Grafem této funkce je dvourozměrný povrch. Je zde tedy podobnost s povrchem Země. A stejně tak můžeme definovat rozvodí a povodí[4]. Každé povodí potom udává jeden segment. Tato myšlenka je sice vcelku jednoduchá, ale implementace algoritmu je složitější.

Jednou z možností je postupné testování všech bodů. Pro každý hledáme cestu „toku řeky“ do lokálního minima. Povodí je potom dáno všemi body, ze kterých nalezneme cestu do stejného minima. Problém tohoto řešení je ve výpočetní složitosti a v nespojitosti povrchu digitálních dat.

Druhou možností je postupovat podle výšky. Nejprve seřadíme body podle jejich výšek. Potom postupně „zaplavujeme“ od minim. Procházíme jednotlivé úrovně od nejnižších. V tom případě buďto nalezneme body nového lokálního minima a označíme je nebo najdeme body, které sousedí s již nalezeným lokálním minimem. Pokud sousedí v dané vrstvě bod pouze s jedním minimem, tak se jedná o povodí tohoto minima, pokud dojde v dané vrstvě ke spojení dvou minim, tak bod náleží bližšímu povodí.

[4]Povodí je oblast, ze které voda odtéká do jedné konkrétní řeky či jezera. Hranice mezi dvěma povodími se nazývá rozvodí. Citováno z [WP10]

Segmentace rozvodím může mít za výsledek příliš jemnou nebo naopak hrubou segmentaci. Proto existuje skupina metod s dodatečnou úpravou segmentů.

Segmentace pomocí trixelů

Protože každý z přístupů hledání segmentů – jak podle hran, tak podle oblastí – má své výhody i nevýhody, vzniká myšlenka, zda by nebylo vhodné použít obě informace v jednom algoritmu pro zlepšení možností segmentace. Tímto se zabývá práce [PS06]. Autoři zde definují jako hlavní jednotku trixel, což je trojúhelník nesoucí informace o jeho barvě. Díky tomu máme informaci jak o hraně, tak o oblasti.

Samotný algoritmus nejprve provede detekci hran za pomoci Cannyho algoritmu. Tím se vytvoří sada omezení pro triangulaci. Provádí se takzvaná triangulace s omezením, kdy všechny vrcholy trojúhelníků musí vždy ležet na některé z nalezených hran. Pomocí metody Monte Carlo (výběr několika náhodných bodů) se vypočte barva trojúhelníku – vzniká trixel. Hrany jednotlivých trixelů jsou potom oddělovače nebo spojky oblastí. Pokud hranu vynecháme, tak trixely vytvářejí spojenou oblast. Naopak pokud ji ponecháme, vytváříme hranu oblasti. Pomocí kritérií homogenity potom postupně dochází ke spojování sousedních trixelů. A vzniká tak graf jednotlivých skupin. Tento graf nese informaci podobnou kostře oblasti. Segmenty potom získáme pomocí algoritmů procházení grafem. Barvy trixelů nám navíc poskytují informaci o průměrné barvě tohoto segmentu.

2.3 Metody porovnání obrazu

Metody porovnání obrazu zjišťují, jsou-li si dva obrazy podobné. Podobnost může být dána mnoha různými faktory. V nejjednodušším případě se jedná o pouhé porovnání hodnot bodů obrazu. Složitější metody se snaží porovnání založit na parametrech, které jsou důležité pro pozorovatele. Jiné dokonce zkoumají struktury a objekty obsažené v porovnávaných obrazech. Výsledkem porovnání bývá číselná hodnota. Její interpretace závisí na jednotlivých metodách. Pro každou uvedenou metodu je vždy popsán její výstup.

Pro zjednodušení budeme porovnávání vysvětlovat na stejně velkých obrazech. V praxi je toto porovnání poměrně časté například u ztrátové komprese, kdy porovnáváme původní a komprimovaný soubor. V tom případě nám porovnání poskytuje hodnotu udávající, k jak velké ztrátě dat při kompresi došlo.

2.3.1 Rozdíl hodnot a MSE

Základní metodou, která nás napadne pro porovnání obrazu, je provedení rozdílu hodnot obou obrazů na stejných pozicích. Tuto myšlenku lze matematicky formulovat následujícím zápisem:

$$ERR = \frac{1}{mn} \sum_{x=0}^{m-1} \sum_{y=0}^{n-1} |I_1(x,y) - I_2(x,y)|,$$

kde I_1 a I_2 jsou porovnávané obrazy o rozměrech $m \times n$. Hodnota ERR je potom výslednou hodnotou porovnání. Pokud je hodnota nulová, jsou obrazy shodné. Čím je tato hodnota větší, tím se obrazy vzájemně více liší.

Většinou se místo jednoduchého rozdílu hodnot využívá takzvaná *střední kvadratická chyba*. Někdy je tato metoda označována jako MSE podle anglického „mean squared error“. Jedná se o stejný princip jako v minulém příkladu, pouze se počítá součet přes kvadrát – druhou mocninu:

$$MSE = \frac{1}{mn} \sum_{x=0}^{m-1} \sum_{y=0}^{n-1} (I_1(x,y) - I_2(x,y))^2$$

Tato metoda je použitelná například pro měření poškození obrazu způsobeného ztrátovou kompresí, jak bylo uvedeno v předchozí části. Pro porovnání dvou různých obrazů není tato metoda příliš vhodná. Například pokud budeme porovnávat černobílý obraz se stejným obrazem posunutým o jeden bod, bude tato metoda vracet veliké hodnoty a výsledkem bude konstatování, že obrazy si nejsou podobné.

2.3.2 SSIM

Článek [WBSS04] popisuje vylepšenou metodu porovnání obrazu. Metoda SSIM (z angl. structural similarity) byla sestrojena tak, aby porovnání obrazů lépe odpovídalo lidskému vnímání. Na rozdíl od předchozích metod bere SSIM v úvahu větší okolí bodu. Porovnávají se jednotlivé podoblasti obrazů. Porovnávaná oblast může být zvolena například jako čtvercová oblast o velikosti 8×8. Pro lepší výsledky je možné použít kruhově souměrné okolí 11×11. Index SSIM se počítá většinou pouze pro jasové složky obrazů.

Index podobnosti dvou oblastí x a y je dán vzorcem:

$$SSIM(x,y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)},$$

kde μ je vážený průměr, σ udává váženou varianci (rozptyl) a kovarianci hodnot. Hodnoty C_i jsou dané vztahem $C_i = (k_i L)^2$. L je rozsah vstupních hodnot (často 255) a k_i jsou konstanty. Většinou se používá $k_1 = 0,01$ a $k_2 = 0,03$. Při výpočtu statistik se používají váhy. Vhodné je například Gaussovské rozložení vah.

Index SSIM vychází jako hodnota $< -1, 1 >$. Hodnota 1 znamená, že porovnávané obrazy jsou shodné. Čím je index menší, tím jsou si obrazy méně podobné.

Výsledný index pro celý obraz se potom počítá jako průměr indexů jednotlivých porovnávaných oblastí obrazů. Označuje se jako MSSIM (mean SSIM) a pro obrazy X a Y je definován jako:

$$MSSIM(X, Y) = \frac{1}{M} \sum_{i=1}^{M} SSIM(x_i, y_i)$$

2.3.3 Strukturální porovnání

Další metody porovnání obrazu jsou založené na porovnání obsahu dvou obrazů. Před porovnáním je nejprve nutné provést analýzu a porozumět obsahu obrazu. Základem porozumění obrazu bývá segmentace. Většinou se jedná o specializované metody, které se zabývají pouze určitou skupinou obrazů. Porozuměním obrazů se zabývá především obor umělé inteligence.

3 Realizační část

Cílem práce je vytvoření uživatelsky jednoduchého systému určeného pro redukci skutečné velikosti dat na disku. Redukce dat je zaměřena na obrazová data. Systému přesto umožní ukládat libovolná data. Redukce obrazu přitom bude pro redukci dat využívat podobnosti jednotlivých uložených obrazových dat.

3.1 Technická dokumentace

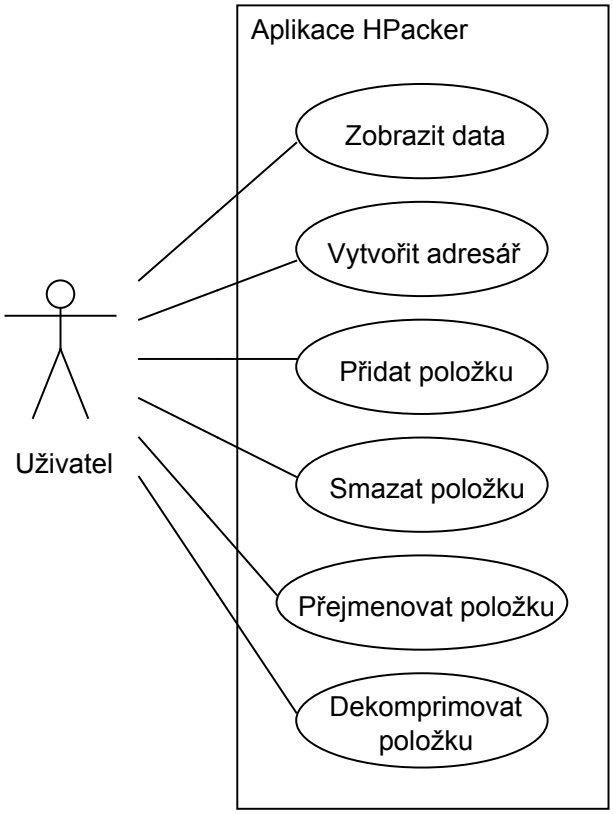
Tato kapitola je zaměřena na strukturu samotné implementované aplikace. Je zde uvedena specifikace požadavků, architektura aplikace, základní datové struktury a podobně.

3.1.1 Specifikace požadavků

Funkční požadavky

Základní funkční požadavky jsou popsány za pomoci UML[1]. Na obrázku 3.1 je zobrazen diagram případů užití. Následují diagramy s popisem jednotlivých případů užití. Protože v celé aplikaci vystupuje jenom jeden aktér, není v diagramech případů užití uveden.

[1]UML (Unified Modeling Language) je univerzální jazyk pro vizuální modelování systémů (definice z [AN07]).



Obrázek 3.1: Diagram případů užití.

Případ užití: **Zobrazit data**
ID: 1
Stručný popis: Vhodným způsobem zobrazí data aplikace. Při zobrazení je možné změnit perspektivu zobrazení.
Vstupní podmínky: • Musí existovat data. • Minimální množinou dat je kořenový adresář aplikace.
Hlavní scénář: 1. Zobrazení dat aplikace. 2. Umožnění interakce s uživatelem a spouštění dalších případů užití.
Výstupní podmínky: Žádné.
Alternativní scénáře: Žádné.

Obrázek 3.2: Případ užití zobrazit data.

Případ užití: **Vytvořit adresář**
ID: 2
Stručný popis: Vytvoří nový adresář.
Vstupní podmínky: • Známe rodičovský adresář, ve kterém se bude vytvářet nový adresář.
Hlavní scénář: 1. Uživatel zadá název nového adresáře. 2. Aplikace vytvoří a uloží nový adresář.
Výstupní podmínky: • Aplikace zobrazuje data včetně nově vytvořeného adresáře.
Alternativní scénáře: 1.a) Uživatel zadá název, který se shoduje s názvem existujícího adresáře v rámci rodičovského adresáře. 1.b) Uživatel je o chybě informován a případ užití končí neúspěchem.

Obrázek 3.3: Případ užití vytvořit adresář.

Případ užití: **Přidat položku**
ID: 3
Stručný popis: Vloží soubor nebo adresář do systému. V případě adresáře se vloží celá jeho struktura. Všechny složky a soubory, které obsahuje.
Vstupní podmínky: • Známe rodičovský adresář, do kterého vkládáme data.
Hlavní scénář: 1. Uživatel vybere položku, kterou chce do systému vložit. 2. Aplikace vytvoří a uloží položky do systému. 3. Aplikace vybere nejvhodnější kompresní metody pro data vkládaných souborů a uloží je pomocí zvolených metod.
Výstupní podmínky: • Aplikace zobrazuje data včetně nově vytvořených položek.
Alternativní scénáře: 2.a) Uživatelem zvolená data se nepodařilo uložit. Např. protože položka se stejným jménem již v rodičovském adresáři existuje. 2.b) Uživatel je o chybě informován a případ užití končí neúspěchem. Pokud bylo vkládáno více položek najednou, není vložena ani jedna z nich.

Obrázek 3.4: Případ užití přidat položku.

Případ užití: **Smazat položku**
ID: 4
Stručný popis: Odstraní soubor nebo adresář ze struktury.
Vstupní podmínky: • Známe položku, kterou chceme smazat.
Hlavní scénář: 1. Testuje se, zda se na mazané soubory neodkazují jiné soubory. 2. Pokud existuje závislost a je možné ji odstranit (např. úpravou struktury dat), bude odstraněna. 3. Pokud existuje závislost a není možné ji odstranit, bude soubor odstraněn pouze ze struktury a uživatel o tom bude informován. 4. Pokud neexistuje závislost, bude soubor odstraněn. 5. Budou odstraněny všechny zvolené adresáře a jejich podadresáře.
Výstupní podmínky: • Aplikace zobrazuje strukturu bez odstraněných položek.
Alternativní scénáře: Žádné.

Obrázek 3.5: Případ užití smazat položku.

Případ užití: **Přejmenovat položku**
ID: 5
Stručný popis: Přejmenuje soubor nebo adresář ve struktuře.
Vstupní podmínky: • Známe položku, kterou chceme přejmenovat.
Hlavní scénář: 1. Zobrazí se vstupní okno pro zadání nového názvu položky. 2. Uživatel zadá nové jméno položky a potvrdí volbu. 3. Položka se přejmenuje.
Výstupní podmínky: • Aplikace zobrazuje strukturu s novým názvem položky.
Alternativní scénáře: 2.a) Uživatel stornuje úpravu jména. 2.b) Případ užití končí a položka není přejmenována. 3.a) Položka se stejným názvem ve stejném adresáři již existuje. 3.b) Uživatel je o chybě informován a položka není přejmenována.

Obrázek 3.6: Případ užití přejmenovat položku.

Případ užití: **Dekomprimovat položku**
ID: 6
Stručný popis: Dekomprimuje data včetně jejich struktury na uživatelem zvolené místo.
Vstupní podmínky: • Známe položku, kterou chceme dekomprimovat. • Známe umístění, kam se mají data dekomprimovat.
Hlavní scénář: 1. Všechny zvolené soubory a adresáře včetně jejich obsahu jsou dekomprimovány na zvolené místo. 2. Po dekompresi data v systému stále zůstávají.
Výstupní podmínky: • Data jsou dekomprimována na zvoleném umístění. • Data jsou uložena v systému.
Alternativní scénáře: 2.a) Při dekompresi došlo k problémům. 2.b) Uživatel je upozorněn na vzniklou chybu.

Obrázek 3.7: Případ užití dekomprimovat položku.

Ostatní požadavky

Tato kapitola popisuje doplnění funkčních požadavků a požadavky nefunkční.

Funkčnost

- Aplikace musí pro kompresi obrazu využívat podobnosti jednotlivých uložených obrazových dat.
- Aplikace nesmí svévolně dekomprimovat a znovu komprimovat soubory při použití ztrátové komprese, protože by se snižovala kvalita uložených dat.

Použitelnost

- Aplikace musí být velmi jednoduše ovladatelná a intuitivní, aby byla použitelná širokým spektrem uživatelů.
- Aplikace bude obsahovat dva dokumenty. Jeden určený běžným uživatelům a druhý, který bude obsahovat podrobnější popis aplikace včetně možností rozšiřitelnosti.

Robustnost

- Pokud je aplikace správně nastavena[2], tak nesmí docházet k pádům aplikace z důvodu uživatelské interakce.
- Pokud dojde při běhu aplikace k problémům, tak musí být uživatel vhodným způsobem informován.
- Všechny závažné problémy aplikace musí být zaznamenány do systémového logu.
- Aplikace bude pouze jednouživatelská a může běžet pouze jednou nad jedinou databází.

[2]Správným nastavením je myšlena především konfigurace a instalace databáze.

Výkon

- Při standardním použití aplikace by mělo docházet k rychlé odezvě. Maximálně v řádech 2-5 sekund.
- Pokud by byla odezva pomalejší, např. při kompresi nebo dekompresi většího množství dat, je nutné zobrazit hlášení o průběhu prováděné operace nebo zobrazit uživateli upozornění, že uvedená operace bude trvat delší dobu.
- V paměti se musí držet vždy minimální množství dat. Po uložení dat do systému by měly být odstraněny z paměti.

Rozšiřitelnost

- Aplikace musí umět pracovat s databází HSQL.
- Aplikace by měla podporovat možnost použití jiné databáze[3]. Přechod na jinou databázi není jen uživatelskou nebo konfigurační změnou[4].
- Aplikace musí být jednoduše lokalizovatelná. Minimálně musí být lokalizována do češtiny a angličtiny.
- Aplikace by měla být jednoduše rozšiřitelná o další metody komprese.
- Aplikace může umožňovat rozšíření o další perspektivy.

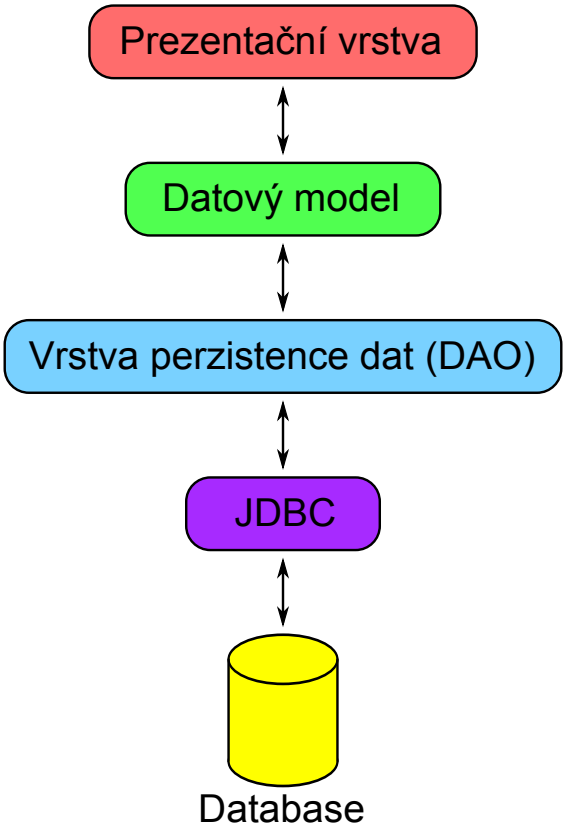
Běhové prostředí

- Aplikace musí bezproblémově fungovat nad operačním systémem Windows. Zejména se jedná o verze Windows XP a Windows 7.

[3]Podpora by se měla vztahovat především pro databáze poskytující JDBC ovladač.
[4]Minimálně je nutná revize skriptů pro vytvoření struktury databáze.

3.1.2 Koncepce aplikace

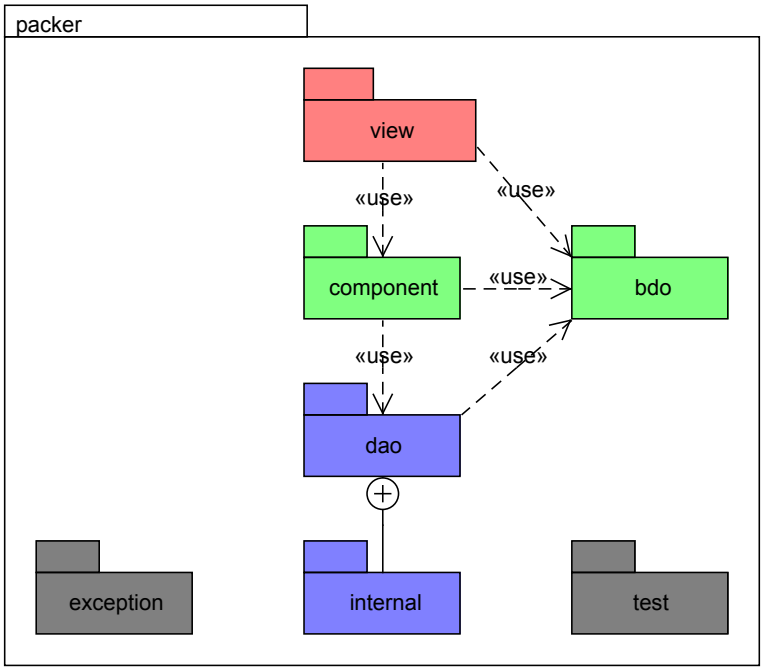
Jedná se o jednouživatelskou desktop aplikaci. Pro implementaci byla zvolena technologie Java. Technologii Java jsem zvolil kvůli její dobré znalosti, podpoře a velmi dobré dokumentaci. Aplikace je implementována jako třívrstvá aplikace (viz Obrázek 3.8). Následně bude popsán stručný význam jednotlivých vrstev. Dále bude následovat podrobnější pohled na jednotlivé vrstvy a jejich komponenty. Celá koncepce staví na komponentovém vývoji. Tato kapitola odkazuje na UML diagramy, které jsou poměrně velké, a proto byly umístěny do přílohy. Diagramy obsahují pouze nejdůležitější části. Pro kompletní popis je vytvořena javadoc dokumentace.



Obrázek 3.8: Třívrstvá aplikace.

Balíková struktura

Aplikace je rozdělena do balíkové struktury. Celá balíková struktura je zobrazena na obrázku 3.9. Hlavním balíkem je `cz.zcu.fav.hofhans.packer`, který obsahuje všechny základní části aplikace. Z uvedeného obrázku komponent jsou patrné vazby mezi jednotlivými balíky. Ty korespondují s modelem třívrstvé aplikace.



Obrázek 3.9: Balíková struktura aplikace.

Balík `view` obsahuje třídy prezentační vrstvy. Datový model reprezentují balíky `component` a `bdo`, kde komponenty tvoří aplikační logiku a bdo[5] jsou určeny pro přenos dat. Jejich účel je patrný i z obrázku komponent a jejich vazeb.

Vrstva prezistence dat je uložena v balíku `dao` a `dao.internal`. Balík `dao` obsahuje rozhraní a `dao.internal` obsahuje vnitřní nahraditelnou implementaci tohoto rozhraní. Účel zbylých dvou balíků `exception` a `test` je zřejmý z jejich názvu. Jedná se o balík obsahující základní výjimky používané aplikací a pomocný balík pro implementaci testů.

[5]Business data object – datový objekt, třídy používané pro přenos dat. Bývají také označovány jako „value object“ nebo „POJO“ (Plain Old Java Object).

Rozšiřitelnost a konfigurace

Většina jednotlivých částí by měla být nahraditelná a rozšiřitelná. Nové komponenty jsou do systému integrovány za pomoci konfigurace a reflexe. Jednoduše lze přidávat například další metody komprese a lokalizaci.

Celá aplikace je konfigurovatelná pomocí `properties` souborů. Jedná se o jednoduché soubory s obsahem – klíč = hodnota. `Properties` soubory byly zvoleny, protože se jedná o jednoduchý formát jak z pohledu struktury dat, tak práce s ním, a k požadavkům konfigurace plně dostačuje. Pokud by byla potřeba složitější strukturovaná konfigurace, byl by vhodnější formát XML nebo YAML.

Databáze

Díky JDBC[6] by aplikace měla být nezávislá na použité databázi. Ve skutečnosti určitá závislost existuje. Týká se to především konfigurace databáze a vytvoření její struktury.

Pro základní použití aplikace byla zvolena databáze HSQLDB[7]. Byla vybrána ze dvou hlavních důvodů. Za prvé se jedná o aplikaci implementovanou za pomoci Java technologie. Jedná se o stejnou technologii, na které je stavěna aplikace, což je výhodou. Druhým důvodem je její jednoduchost. Celá databáze je uložena v několika málo souborech a není nutná její instalace. Díky tomu je možné uložená data velmi jednoduše přenášet nebo zálohovat.

Vrstva perzistence dat

Tato vrstva zajišťuje ukládání dat. V této vrstvě bude probíhat samotná komprese. Vrstva perzistence dat zajišťuje práci s transakcí na úrovni databáze. Je vytvořeno rozhraní pro vyšší vrstvy, které umožňuje načítat a ukládat data. Toto rozhraní je vytvořeno v balíku `cz.zcu.fav.hofhans.packer.dao`. Rozhraní bude podrobně rozebráno později.

Aplikace bude obsahovat výchozí implementaci tohoto rozhraní, která bude ukládat data za pomoci rozhraní JDBC do perzistentní databáze. Díky použití JDBC by změna databáze neměla znamenat žádný zásah do implementace[8]. Pokud by bylo potřeba změnit cíl ukládání razantnějším způsobem, například tak, že nelze použít rozhraní JDBC,

[6] Java Database Connectivity – Java rozhraní pro přístup k databázi.
[7] Podrobnější informaci lze nalézt v popisu reference [HSQLDB].
[8] Problém by mohl být, kdyby databáze nepodporovala všechny funkčnosti rozhraní JDBC.

potom bychom museli změnit implementaci prakticky celé této vrstvy. Výhodou je, že díky rozhraní se zbytek aplikace nemění.

Datový model

Vrstva datového modelu poskytuje dvě důležité věci. Za prvé obsahuje třídy pro uchování a přenos informace. Za druhé obsahuje rozhraní pro úpravu datového modelu. Jinými slovy obsahuje aplikační logiku. Podrobný popis datového modelu bude vysvětlen později.

Z pohledu prezentační vrstvy poskytuje datový model data pro zobrazení a metody pro jejich úpravu.

Prezentační vrstva

Stará se o vhodné zobrazení datového modelu. Pro implementaci prezentační vrstvy bude použito Java Swing[9]. Základem je jednoduché okno, které zobrazuje menu, stavový panel (status bar) a hlavní okno aplikace. Menu slouží k výběru základních funkčností. Stavový panel zobrazuje stav aplikace – např. progress bar. Hlavní okno zobrazuje vhodným způsobem aplikační data a pomocí kontextových menu umožňuje interakci aplikace s uživatelem.

3.1.3 Datové struktury

Tato část popisuje datové struktury aplikace. Popisuje jednotlivé použité třídy, databázovou strukturu a příklady použití jednotlivých objektů. Všechny třídy mají anglické názvy, protože implementace je psána anglicky. Většina tříd začíná předponou `Packer`. Důvodem je jednodušší identifikace vlastních tříd a zmenšuje se problém s kolizí názvů tříd[10].

[9] Bylo zvoleno na základě nejlepší znalosti. Riziko spojené s použitím neznámých knihoven bylo příliš velké.

[10] Pokud bychom vytvořili například třídu `File` byl by název stejný s třídou jádra Javy `java.io.File` a byla by potřeba používat plně kvalifikované jméno. Navíc by stejné označení tříd bylo velmi nepřehledné.

Základní třídy modelu

Systém spravuje strukturu dokumentů. Pro vytvoření struktury slouží adresář, který je reprezentován třídou `PackerFolder`. Dokument (soubor) reprezentuje třída `PackerFile`. Soubory a adresáře jsou souhrnně označovány jako položka. Položka je jejich společným předekem s názvem `PackerItem`. Soubory mohou obsahovat data, ale ty se v paměti drží pouze po nezbytně nutnou dobu. Poté jsou nastaveny na `null` a před následujícím použitím je nutné je znovu načíst z databáze.

Každý soubor navíc obsahuje typ komprese popsaný třídou `CompressionType` a parametry souboru, podle kterých lze vybrat nejvhodnější komprimační metodu. Parametry souboru jsou reprezentovány třídou `CompressionParams`. Parametry lze jednoduše rozšiřovat podle potřeby aplikace. Třída `ComressionType` obsahuje informace o všech možných typech komprese načtených z konfigurace. Typ komprese obsahuje název[11] a třídu, která se stará o perzistenci dat – implementace komprese/dekomprese a uložení/ /načtení dat.

Kompletní přehled datových tříd určených pro uchování a přenos informace je znázorněn v diagramu A.2. Všechny třídy, uvedené v diagramu, jsou uloženy v balíku `cz.zcu.fav.hofhans.packer.bdo`. Balík navíc obsahuje třídu `Context`, určenou pro uchování aktuálního stavu aplikace a pro případnou komunikaci mezi jednotlivými komponentami. Kontext aplikace se po skončení ukládá, aby byl zachován stav aplikace do příštího spuštění.

Dále systém obsahuje modely určené pro práci nad těmito objekty. Jedná se většinou o třídy, které implementují rozhraní Java Swing a je možné je jednoduše zobrazovat pomocí standardních komponent. Pro práci nad strukturou adresářů se využívá třída `PackerFolderTreeModel`, která obsahuje položky `PackerFolderTreeNode`. Obsah jednoho adresáře je reprezentován třídou `PackerItemListModel`. Modely jsou dokumentovány na diagramu A.3.

Rozhraní pro přístup k datům

Rozhraní pro přístup k datům obsahuje dvě pomocné třídy a rozhraní pro samotný přístup k datům. Kompletní diagram rozhraní je na diagramu A.4.

Rozhraní je definováno pomocí `FileDao`, `FolderDao`, `CompressionDao` a `TXManager`. První dvě rozhraní definují metody pro ukládání a načítání strukturních položek. Roz-

[11] Respektive kódem, který bude lokalizován.

hraní `CompressionDao` musí implementovat všechny použité kompresní metody. Definuje metody pro práci s daty jednotlivých souborů. Asi nejzajímavější metodou je metoda `evaluateCompression`, která vrací hodnotu $< 0,1 >$ jako vhodnost komprese. 0 znamená, že komprese je nevhodná a naopak hodnota 1 udává nejlepší možnou kompresní metodu pro uvedený soubor. Výjimkou je návratová hodnota menší než 0, která udává, že uvedený způsob komprese nelze použít. Pomocí této metody se vyhodnotí nejlepší možná komprese pro daný soubor. Posledním rozhraním je `TXManager`. Jedná se o velmi jednoduchou podporu transakcí. Za transakce je v tomto případě odpovědná vrstva datového modelu (aplikační vrstva), která využívá `TXManager` pro řízení transakcí.

Pomocné třídy jsou `PackerDaoFactory` a `ConnectionManager`. Třída `PackerDaoFactory` je třída obsahující pouze statické tovární metody. Nelze vytvořit instanci této třídy. Používá se pro získání instancí implementující uvedená rozhraní. Třída `ConnectionManager` je určena pro získání a práci s připojením do databáze. Kromě inicializace by měl být používán výhradně vrstvou pro přístup k datům.

Navíc je v odkazovaném diagramu uvedena třída `CompressionType` z balíku základního modelu. Tato třída načítá pomocí reflexe všechny kompresní metody, které jsou definovány rozhraním `CompressionDao`. Každý soubor je tak spojen s kompresní metodou, jaká se používá pro uložení dat. Pro vytvoření nové kompresní metody stačí implementovat rozhraní `CompressionDao` a přidat její název a plně kvalifikované jméno implementované třídy do konfigurace.

Databázové schéma

Databázové schéma je zobrazeno na obrázku A.1. Každá kompresní metoda je uložena v samostatné tabulce a odkazuje na soubor, jehož data ukládá. Je tomu tak ze dvou důvodů. Za prvé má každá kompresní metoda vlastní sadu parametrů, které se liší. Druhým důvodem je rozšiřitelnost, kdy při rozšíření aplikace o novou kompresní metodu vzniká nová tabulka, kam se budou data ukládat. Takže každá implementace třídy `CopressionDao` spravuje svoji tabulku, kam ukládá data. Nový typ komprese může navíc zvolit odlišný způsob ukládání a ukládat svá data např. do vlastních souborů.

Model prezentační vrstvy

Prezentační vrstva aplikace staví na návrhové vzoru MVC[12]. Základem je rozhraní `PackerView`, které definuje metody povinné pro všechny prezentační komponenty, a abstraktní třída `PackerAbstractView`, která dané rozhraní implementuje a obsahuje obecné metody použitelné jednotlivými prezentačními komponentami. Komunikace mezi jednotlivými částmi této vrstvy se provádí za pomoci návrhového vzoru posluchač (Listener). Pro udržení celkového stavu aplikace a přenos libovolné informace mezi perspektivami se používá třída `Context` balíku `bdo`. Třída `Controller` je vstupním bodem celé aplikace a jejím účelem je správa jednotlivých komponent. Všechny uvedené prvky jsou zobrazeny na obrázku A.5. Původní myšlenkou bylo umožnění více přepínatelných perspektiv. Díky uvedenému rozhraní by to mělo být možné, ale protože současná implementace obsahuje pouze jednou perspektivu, není tato možnost implementována.

3.2 Realizace komprese

3.2.1 Základní myšlenka

Základní myšlenkou celé práce je hledání podobnosti nikoliv přímo v obrazových datech, ale v jejich Fourierově obrazu. Celá myšlenka vychází z JPEG komprese. Celý základní proces komprese je stejný. Nejprve se obraz převede do barevného prostoru YC_rC_b. Poté se provádí segmentace. Obraz se jednoduše dělí na bloky velikosti 8×8 bodů. Následuje transformace z jasových hodnot jednotlivých složek do frekvenčního prostoru - diskrétní kosinová transformace. Koeficienty získané transformací jsou následně kvantovány.

Nyní máme bloky velikosti 8×8, které můžeme reprezentovat jako řídké matice s dominantním levým rohem. Především v barevných složkách obsahují matice pouze několik nenulových členů. V této chvíli můžeme hledat podobné nebo stejné matice. Uložení hodnot bloku můžeme nahradit odkazem na stejnou nebo podobnou matici.

Stejně jako u JPEG komprese se stejnosměrné složky, hodnoty z levých horních rohů matice, ukládají zvlášť. Bloky se ukládají opět stejným způsobem jako v JPEG kompresi za pomoci entropického kódování v „cik cak“ pořadí. Navíc musíme pro každý obraz uložit pole odkazů na jednotlivé bloky. Spolu s bloky je možné ukládat i informace, které nám umožní rychlejší vyhledávání podobných bloků.

[12]Model-view-controller je softwarová architektura, která rozděluje datový model aplikace, uživatelské rozhraní a řídicí logiku do tří nezávislých komponent tak, že modifikace některé z nich má minimální vliv na ostatní (citováno z [WM10]).

Následující kapitoly popisují jednotlivé zkoumané části kompresního algoritmu, které jsou odlišné od klasické JPEG komprese. Každá kapitola obsahuje výsledky měření provedených v dané oblasti. Při kvantizaci byly využity standardní kvantizační matice JPEG komprese násobené koeficientem $0,5$. Není-li uvedeno jinak, pod pojmem blok budeme uvažovat blok dat po diskrétní transformaci a kvantizaci.

3.2.2 Stejné bloky

Nejprve nebudeme uvažovat žádnou podobnost a vybereme pouze stejné bloky. Jedná se v podstatě o test, jaká je pravděpodobnost, že obraz obsahuje stejné bloky. Testováno bylo celkem 6 různých obrazů. Jejich náhled je na obrázku 3.10. První tři jsou malé obrazy ve formátu PNG nebo TIFF, druhá trojice testovaných obrazů jsou obrazy větších rozměrů ve formátu JPEG. Detailní náhled jednotlivých obrazů je možné nalézt v příloze jako obrazy A.6 až A.11.

(a) Lena (b) Baboon (c) Fruits

(d) Rocks (e) Lake (f) Cows

Obrázek 3.10: Použité testovací obrazy.

Při hledání stejných bloků se procházejí všechny bloky obrazu. Nejprve se prochází bloky jasové složky Y a následně barevné složky C_r a C_b. Každý unikátní blok musíme uložit alespoň jednou, proto musíme při hledání sestavovat databázi unikátních bloků. Hledání shodných bloků v barevných složkách je proto trochu zvýhodněno oproti složce jasové, která je prohledávána jako první. Jistý vliv to má především na menší testované výsledky, ale tento vliv je poměrně malý a na výsledcích měření tento fakt nic nemění.

Obraz		Lena	Baboon	Fruits	Rocks	Lake	Cows
Bloků	Celkem	12288	12288	12288	181476	147456	147456
	1 složka	4096	4096	4096	60492	49152	49152
Četnost stejných bloků	Celkem	7307	2678	8066	133498	111499	98110
	Y-složka	285	0	571	13455	13221	15
	C_r-složka	3435	1487	3842	60442	49149	49060
	C_r-složka	3587	1191	3653	59601	49129	49035
Pravděpod. stejných bloků v %	Celkem	59,46	21,79	65,64	73,56	75,62	66,54
	Y-složka	6,96	0,00	13,94	22,24	26,90	0,03
	C_r-složka	83,86	36,30	93,80	99,92	99,99	99,81
	C_r-složka	87,57	29,08	89,18	98,53	99,95	99,76

Tabulka 3.1: Tabulka měření počtu stejných bloků v jednom obrazu.

Z tabulky 3.1 je patrné, že velmi záleží na vstupních datech. U obrazů, které obsahují větší množství šumu nebo více ostrých hran, je pravděpodobnost nalezení stejných bloků menší. Větší množství shodných bloků je v barevných složkách. To je dáno dvěma důvody. Za prvé kvůli tomu, že jsou barevné složky více kvantovány (kvantizační matice obsahuje větší hodnoty). Za druhé obrazy ve složkách C_r a C_b jsou více „ploché“ – neobsahují velké hrany.

Obraz		Lena	Baboon	Fruits	Rocks	Lake	Cows
Bloků	Celkem	12288	12288	12288	181476	147456	147456
	1 složka	4096	4096	4096	60492	49152	49152
Četnost stejných bloků	Celkem	8045	3290	8872	141087	114653	98536
	Y-složka	623	2	1024	20568	16349	233
	C_r-složka	3696	1900	3969	60481	49152	49151
	C_r-složka	3726	1388	3879	60038	49152	49152
Pravděpod. stejných bloků v %	Celkem	65,47	26,77	72,20	77,74	77,75	66,82
	Y-složka	15,21	0,05	25,00	34,00	33,26	0,47
	C_r-složka	90,23	46,39	96,90	99,98	100,00	100,00
	C_r-složka	90,97	33,89	94,70	99,25	100,00	100,00

Tabulka 3.2: Tabulka měření počtu stejných bloků v naplněné databázi.

Nyní vzniká otázka, jak se výsledky hledání stejných bloků změní v případě, že hledáme ve větším množství obrazů. Do systému bylo uloženo 14 různých obrazů. Všechny vložené obrazy byly ze stejného archivu jako 3.10d. Jedná se o podobné obrazy stejné nebo podobné velikosti uložené ve stejném formátu (JPEG). Jako patnáctý byl do systému vložen testovací obraz. Test byl proveden opět pro všech šest testovacích obrazů. Výsledek měření je uveden v tabulce 3.2.

Z výsledků hledání stejných bloků v naplněné databázi lze odvodit následující závěry:

1. U menších obrazů došlo k výraznějšímu zvětšení pravděpodobnosti nalezení stejných bloků.
2. V případě méně zašuměných obrazů došlo k nárůstu pravděpodobnosti stejných bloků ve složce Y kolem 10%.
3. U obrazů, které obsahují velké množství hran a šumu, jako jsou 3.10b a 3.10f, nedošlo k žádnému razantnímu zlepšení při hledání shodných bloků. Především ve složce Y se výsledky téměř nemění.

Závěr 1. odpovídá tomu, že malé obrazy obsahují méně bloků, mezi kterými je možno najít shodu. Zlepšení je markantnější na rozdíl od větších obrazů především v barevných složkách, protože u velkých obrazů zde již není velký prostor pro vylepšení výsledků.

Další zvětšování databáze bloků má na hledání shodných bloků pouze velmi malý vliv. Zde byl proveden pouze jeden test, kdy bylo vloženo do systému 54 obrazů ze stejného archivu jako 3.10d a následně byl vložen právě obraz 3.10d. Pravděpodobnost nalezení shodných bloků ve složce Y se zvýšila oproti databázi se čtrnácti obrazy o pouhé $1,7\%$.

3.2.3 Podobné bloky

Předchozí kapitola se zabývala hledáním naprosto shodných bloků. Tato kapitola se zaměřuje na podobnost jednotlivých bloků.

Funkce podobnosti

Abychom mohli hovořit o podobnosti, musíme nejprve definovat metriku, která bude udávat, jak moc nebo málo si jsou dva bloky podobné. Volba metriky vychází z teorie

porovnání obrazu popisovaného v kapitole 2.3. Metriku lze nazývat jako funkci podobnosti.

Jako základní metrika pro porovnání dvou bloků byla zvolena střední kvadratická chyba – MSE:

$$MSE = \frac{1}{mn} \sum_{x=0}^{m-1} \sum_{y=0}^{n-1} \left(B_1(x,y) - B_2(x,y)\right)^2$$

B_1 a B_2 jsou porovnávané bloky. Hodnoty x a y jsou potom souřadnice bodu v bloku.

Druhou alternativou je možnost použít váhovou funkci. Vážená střední kvadratická chyba je potom:

$$MSEW = \frac{1}{mn} \sum_{x=0}^{m-1} \sum_{y=0}^{n-1} \left(B_1(x,y) - B_2(x,y)\right)^2 * v(x,y)$$

V tomto případě předpokládáme, že hodnoty v levém horním rohu matice mají větší váhy, naopak hodnoty v pravém dolním rohu jsou méně významné a mají menší váhy. Váhovou funkci můžeme definovat jako vzdálenost[13] od pravého dolního rohu matice. Pro blok o velikosti 8×8 je váhová funkce:

$$v(x,y) = 14 - x - y$$

Predikce

Celá databáze bloků obsahuje velké množství bloků již pro jeden uložený obraz (řádově desetitisíce). Kdybychom chtěli testovat naprosto všechny bloky, by byl algoritmus velmi časově náročný. Proto při hledání podobných bloků budeme využívat určitou predikci nebo-li heuristiku.

Vhodným ukazatelem je například počet nulových prvků v bloku. Lze předpokládat, že bloky, které jsou podobné, mají přibližně stejný počet nulových prvků. Druhou možností je celková suma hodnot bloku. Podobné bloky by měly mít podobnou hodnotu součtu všech jejich prvků. Při výpočtu sumy je možné využít váhové funkce, která může být definována stejně jako váhová funkce použitá při výpočtu podobnostní funkce. Jednotlivá prediktivní kritéria je možné samozřejmě kombinovat.

Při použití predikce můžeme výrazně zúžit množinu prohledávaných bloků. Na druhou stranu není predikce přesná, a proto při hledání podobných bloků přijdeme o určitou

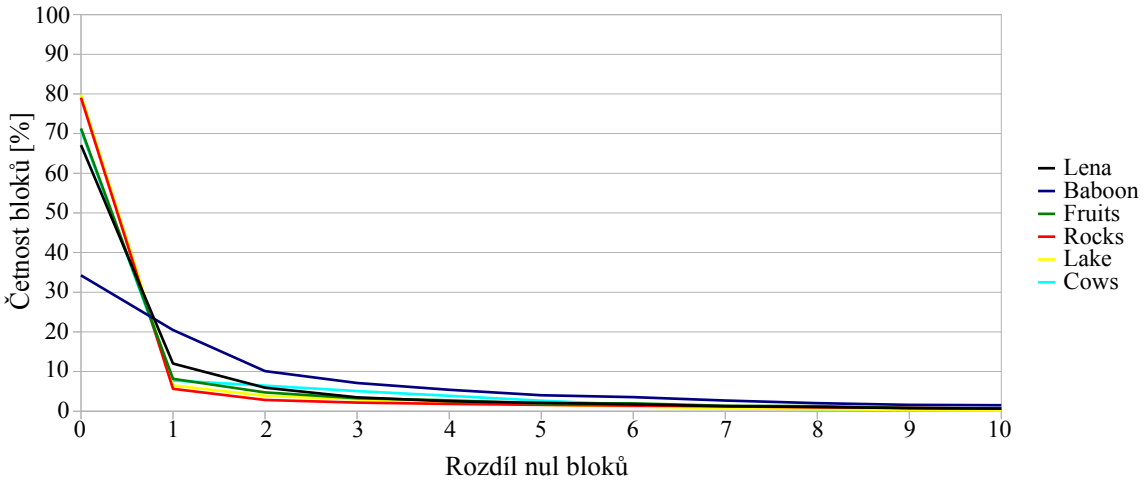
[13]Nejedná se o Euklidovskou vzálenost, ale o vzdálenost Minkovského.

množinu vhodných bloků. Proto je nutné udělat kompromis, kdy dostatečným způsobem zúžíme prohledávanou množinu, ale množina nejpodobnějších bloků, o které přijdeme, je malá.

Měření výsledků

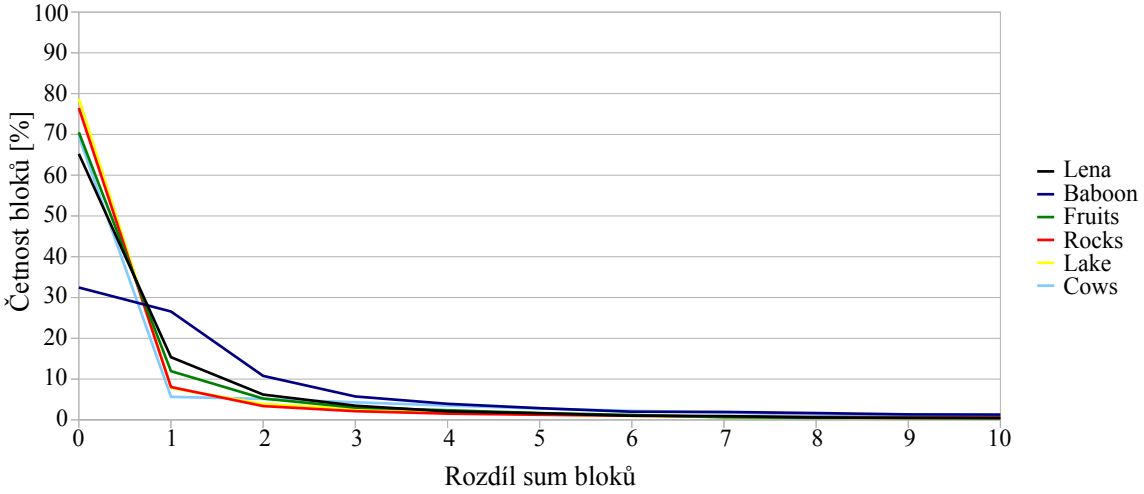
Stejně jako v předchozí kapitole budeme nejprve měřit podobnost v jednom jediném obrazu. Testovací obrazy jsou opět stejné.

Nejprve zvolíme vhodnou metodu predikce pro hledání podobných bloků. Za pomoci metody MSE nalezneme pro každý blok nejpodobnější blok a vypočteme rozdíl nul a sum mezi hledaným blokem a blokem nejpodobnějším. Z naměřených hodnot můžeme sestavit graf relativní četnosti prediktivní funkce.



Obrázek 3.11: Relativní četnost rozdílu nul hledaného bloku s blokem nejpodobnějším.

Byly testovány obě popsané metody predikce. Graf 3.11 udává četnost rozdílu nul bloků a graf 3.12 udává četnost rozdílu sum bloků. Sumy byly počítány bez použití váhové funkce. Z výsledků vychází, že pokud pro obě hodnoty budeme uvažovat maximální rozdíl 5, přijdeme o 2 až 3% nejpodobnějších bloků. Při hledání podobných bloků místo nich pravděpodobně nalezneme jiné podobné bloky s horším výsledkem v porovnání podobnosti nebo se může stát, že jiný podobný blok nenalezneme vůbec. Další testování bude prováděno s predikcí nastavenou pro obě hodnoty na hodnotu 5. To znamená, že na podobnost budeme testovat pouze bloky obsahující maximálně o 5 nul více nebo méně a bloky, jejichž součet hodnot je maximálně o 5 větší nebo menší.

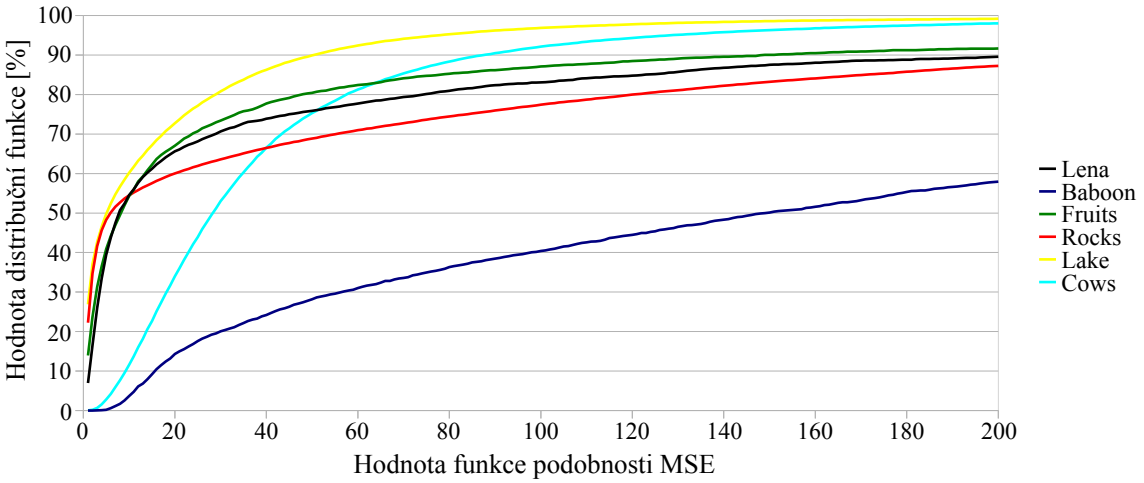


Obrázek 3.12: Relativní četnost rozdílu sum hledaného bloku s blokem nejpodobnějším.

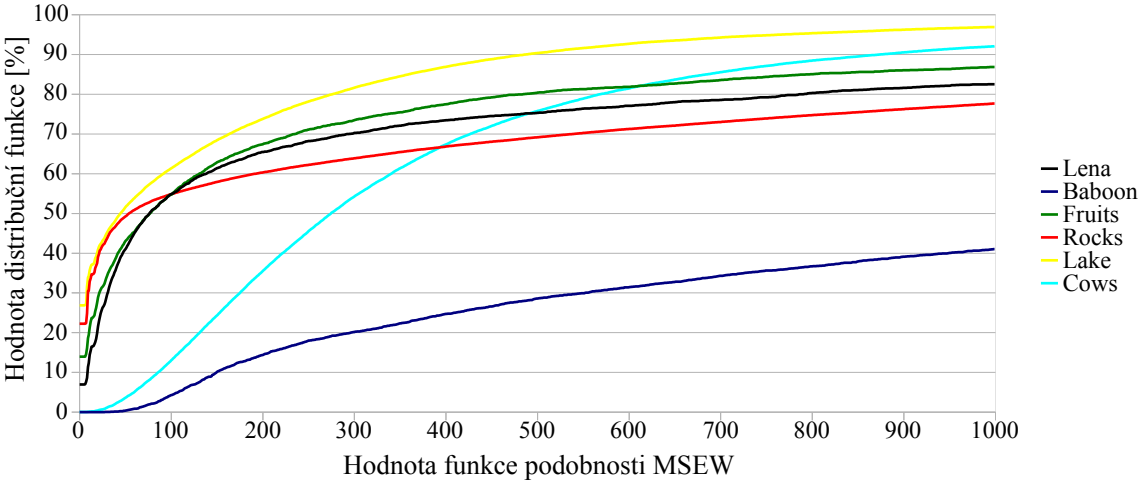
Nyní přejdeme k samotnému hledání podobných bloků. Pro každý blok v testovaném obrazu hledáme blok, který je mu nejpodobnější podle zvolené metriky. Pro obě metriky znamená nejlepší podobnost minimální hodnota funkce metriky. Z naměřených hodnot můžeme získat četnosti výskytu jednotlivých hodnot pravděpodobnostní funkce v testovaném obrazu. Následně z četností můžeme odvodit distribuční funkci. Výpočty budeme provádět v relativních hodnotách – většinou převedených na procenta. Distribuční funkce nám udává, pro kolik procent bloků obrazu lze nalézt podobný blok při zvolené prahové hodnotě. Máme-li např. distribuční funkci $F(x)$ a platí, že $F(10) = 25\%$, tak to znamená, že pokud budeme hledat bloky podobné již uloženým blokům s podobností menší nebo rovnou hodnotě 10, tak jich v obrazu nalezneme celkem 25% z celkového počtu bloků v obrazu.

Měření bylo provedeno pro obě uvedené metriky. Stejně jako pro předchozí měření platí, že jasová složka je lehce znevýhodněna, protože hledání podobných bloků začíná právě u jasové složky obrazu. Naměřené výsledky jasové složky a složek barevných se výrazně liší, proto je budeme studovat odděleně. Graf 3.13 zobrazuje distribuční funkci při použití metriky MSE a graf 3.14 při použití metriky MSEW. Oba grafy zobrazují pouze výsledek pro jasovou složku obrazu.

Z porovnání obou grafů je patrné, že výsledky obou metrik jsou téměř shodné. Největší rozdíl je možné pozorovat pro obraz 3.10f, který obsahuje větší množství šumu. Metrika MSE je jednoduší pro výpočet a navíc větší váha levého horního rohu matice je uvažována již při kvantizaci koeficientů. Proto budeme v této kapitole testovat již pouze metriku MSE. Tato varianta byla zvolena i pro výslednou implementaci.

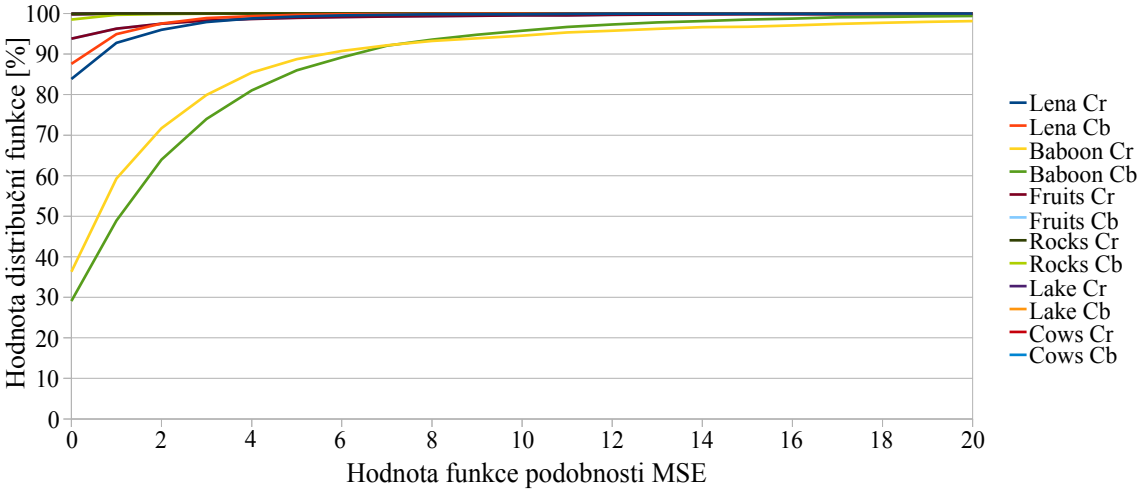


Obrázek 3.13: Distribuční funkce relativní četnosti hodnot podobností pro bloky jasové složky.



Obrázek 3.14: Distribuční funkce relativní četnosti hodnot podobností pro bloky jasové složky – použití váhové funkce.

Podobně jako při hledání stejných bloků platí, že bloky barevných složek jsou si více podobné. Graf 3.15 udává distribuční funkce relativních četností hodnot podobností bloků barevných složek. Je vidět, že pro většinu obrazů začíná funkce u vysokých hodnot (to vyplývá již z měření pravděpodobnosti nalezení stejných bloků). V případě velmi zašuměného malého obrazu, kterým je obraz 3.10b, je vidět, že distribuční funkce začíná mezi 30% až 40%, ale velmi rychle roste.



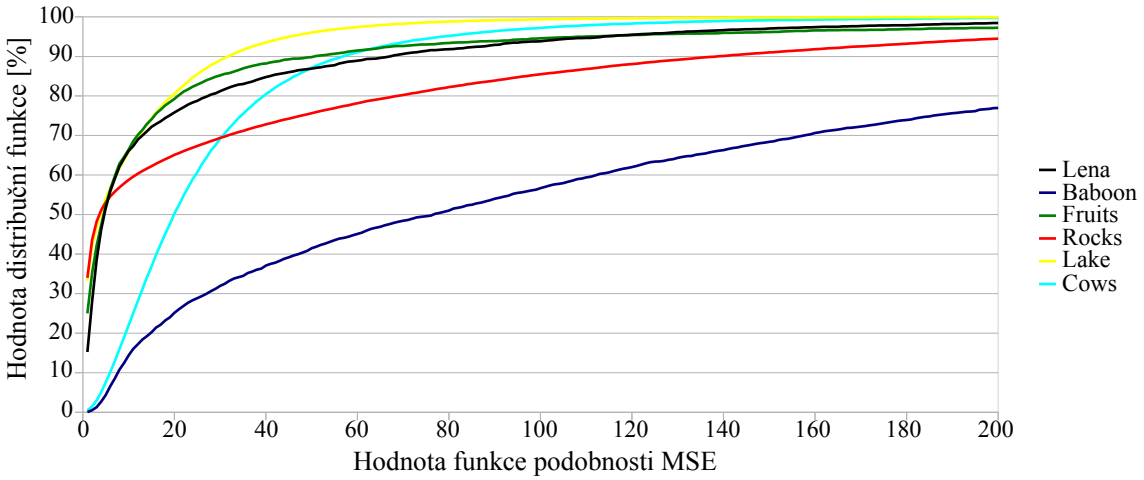
Obrázek 3.15: Distribuční funkce relativní četnosti hodnot podobností pro bloky barevné složky.

Dále je vhodné zjistit, jakým způsobem se změní distribuční funkce v případě naplněné databáze. Do systému bylo opět předem vloženo 14 obrazů. Jednalo se o stejné obrazy jako v kapitole 3.2.2. Jako patnáctý byl vložen testovaný vzorek a opět byla měřena distribuční funkce relativní četnosti hodnot podobností.

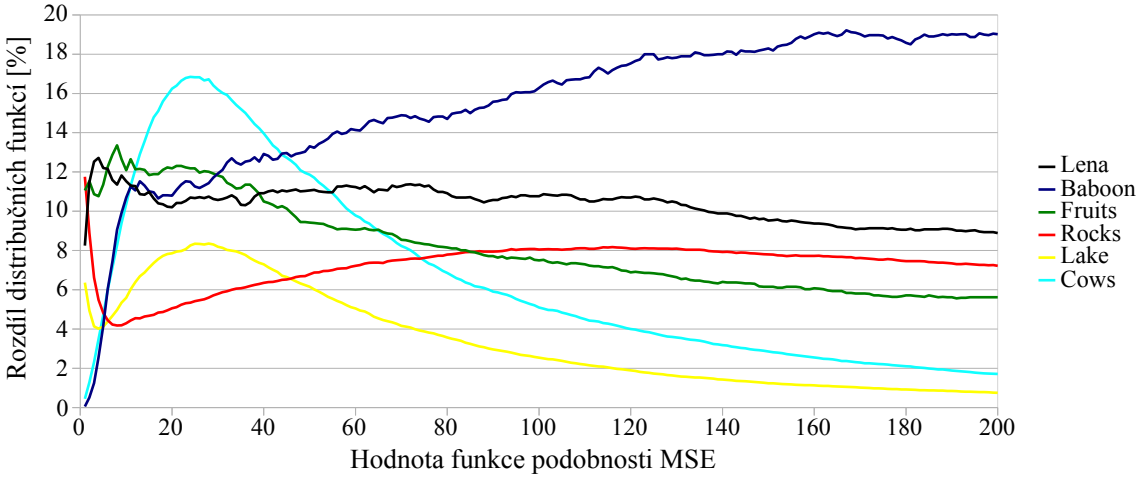
Distribuční funkce pro jasové složky testovaných obrazů je zobrazena v grafu 3.16. Aby byla zřejmá změna hodnot, byl vytvořen graf rozdílu distribučních funkcí s prázdnou a s naplněnou databází. Rozdílový graf je uveden na obrázku 3.17. Jedná se o rozdíl hodnot mezi grafem 3.13 a 3.16.

Z rozdílu hodnot u jasových složek je patrné, že došlo ke zlepšení vyhledávání podobných bloků kolem 10%. Zvláště pro zašuměné obrazy 3.10b a 3.10f došlo k výraznému zlepšení při hledání podobných bloků při nastavení podobnosti MSE v rozmezí 10-40.

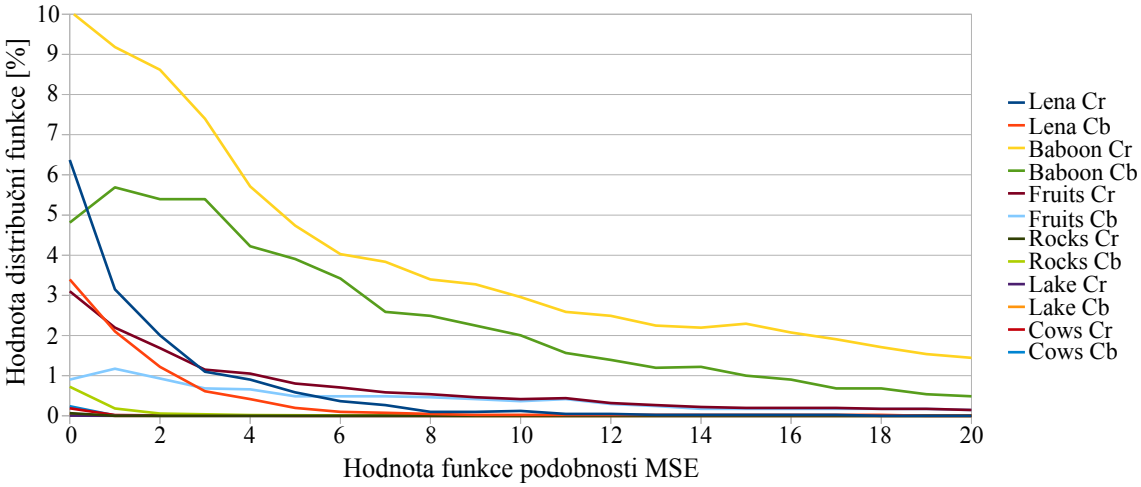
V případě barevných složek máme velmi malý prostor pro zlepšení výsledků, protože již vyhledávání stejných a podobných bloků v jednom obraze nám dává dostatečně dobré výsledky. Pro úplnost výsledků je v grafu 3.18 zobrazena změna distribuční funkce po naplnění databáze pro barevné složky. Změna se projeví především pro zašuměné obrazy, kde je nějaký prostor pro zlepšení. V našem případě se jedná o obraz 3.10b, kde se zlepší především vyhledávání stejných bloků (jak je vidět i z tabulky 3.2).



Obrázek 3.16: Distribuční funkce relativní četnosti hodnot podobností pro bloky jasové složky s naplněnou databází.



Obrázek 3.17: Rozdíl distribučních funkcí pro bloky jasové složky s prázdnou a s naplněnou databází.

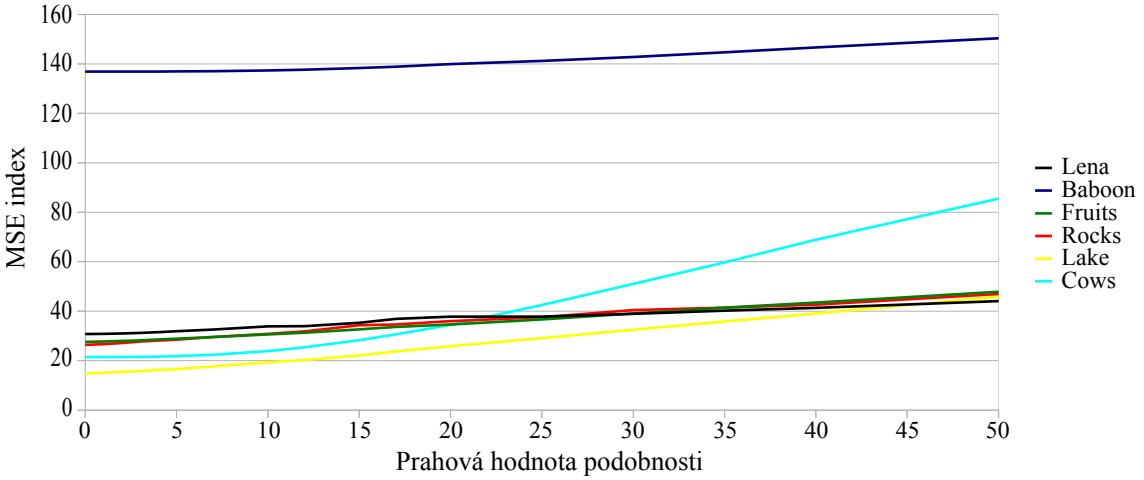


Obrázek 3.18: Rozdíl distribučních funkcí pro bloky barevných složek s prázdnou a s naplněnou databází.

3.2.4 Vliv použití podobnosti na kvalitu obrazu

V této kapitole se budeme zabývat vlivem použití podobnosti na kvalitu obrazu. Kvalitu můžeme měřit za pomoci metod uvedených v kapitole 2.3. Především potom metodou SSIM. Každý testovaný obraz uložíme za pomoci standardního formátu JPEG. Následně provedeme vlastní uložení pomocí unikátních bloků. Porovnání obrazů by mělo být téměř shodné. Poté budeme měnit hodnotu podobnosti pro ukládání bloků.

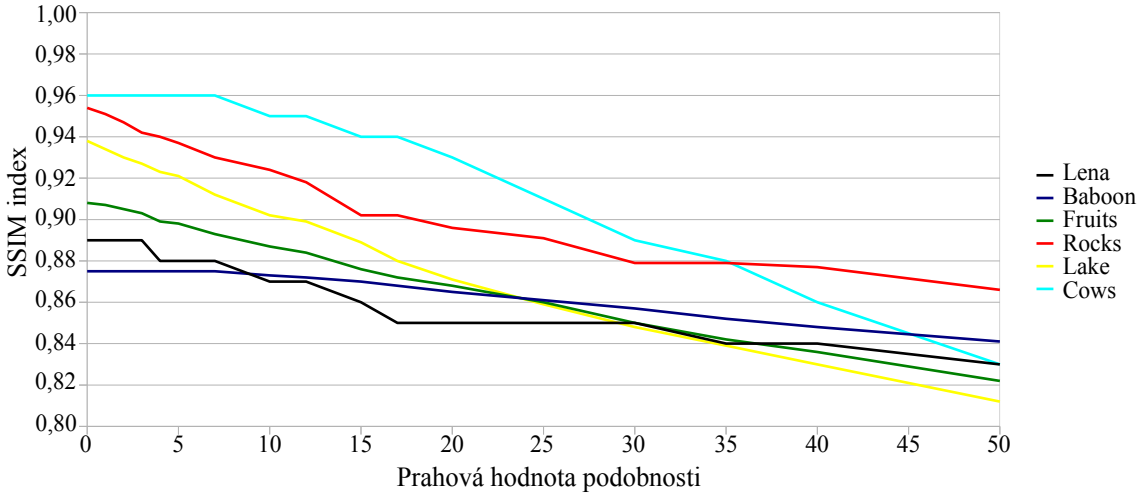
Při hledání podobných bloků budeme podobnost uvažovat pouze v jasové složce obrazu. V barevných složkách budeme vyhledávat pouze bloky zcela shodné. Měření provádíme s prázdnou databází, takže se hledá podobnost pouze v rámci testovaného obrazu.



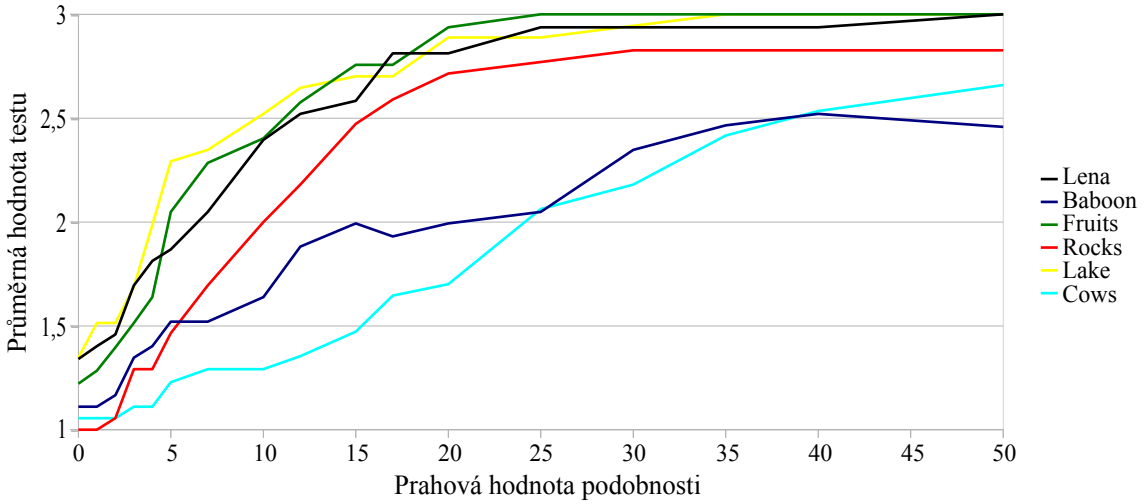
Obrázek 3.19: Měření kvality použitím MSE.

Měřením kvality obrazu v závislosti na prahové hodnotě podobnosti vznikly grafy 3.19 a 3.20. Z uvedených grafů je patrné, že při zvyšování prahové hodnoty se zhoršuje kvalita obrazu. Problémem je, že z naměřených není možné určit, jaká prahová hodnota je ještě přípustná a jaká nikoliv.

Měření pomocí standardních metod porovnání obrazu nám nedává žádné relevantní informace. Proto bylo zapotřebí vyzkoušet jinou metodu. Nejvhodnějším testem je obrátit se přímo na uživatele. Všechny testované obrazy uložené při různé prahové hodnotě podobnosti byly předány vzorku uživatelů. Test provedlo celkem 17 uživatelů. Z toho bylo 9 mužů a 8 žen. Uživatelé měli za úkol ohodnotit obrazy. Hodnocení bylo následující: 1 – nelze poznat rozdíl oproti originálu, 2 – lze pozorovat rozdíl, ale změny nezhoršují kvalitu obrazu, 3 – obraz v této kvalitě nelze akceptovat.



Obrázek 3.20: Měření kvality použitím indexu SSIM.



Obrázek 3.21: Měření kvality vzorkem uživatelů.

Z výsledků byl sestaven graf 3.21. Uvedený graf nám dává celkem dobrou představu o změně kvality v závislosti na použité prahové hodnotě podobnosti. Na obrazy, které jsou více zašuměné jako 3.10b a 3.10f, má prahová hodnota menší vliv. To je způsobeno tím, že v uvedených obrazech je poměrně málo podobných bloků, a proto je méně bloků zkreslených. Podle zvoleného hodnocení je cílem pohybovat se kolem hodnoty 2. Z toho plyne, že při použití prahové hodnoty podobnosti do hodnoty 5 by měl být uživatel s kvalitou obrazu většinou spokojen. Prahová hodnota 10 je pravděpodobně maximální použitelná a větší prahové hodnoty mají zjevný vliv na kvalitu obrazu.

3.2.5 Kódování a uložení hodnot

Jakmile nalezneme unikátní a podobné bloky, ze kterých se obraz skládá, je potřeba tyto bloky vhodným způsobem zakódovat a uložit. Navíc je potřeba uložit reference na bloky, ze kterých se obraz skládá. JPEG komprese využívá obecně entropické kódování. Většina algoritmů je postavena na Huffmanovo kódování. Já jsem využil aritmetické komprese.

Kódování bloků

Každý kvantovaný blok obsahuje 64 celočíselných hodnot uložených na čtyřech bytech. Stejnosměrnou složku budeme kódovat zvlášť, stejně jako to dělá JPEG komprese. Lze předpokládat, že následujících 63 hodnot jsou většinou malé hodnoty se znaménkem. Uvedené hodnoty potřebujeme zakódovat do bytových hodnot.

Součástí dat bloku může být jeho identifikační číslo – ID. V tom případě nejprve zakódujeme identifikátor bloku. V tomto případě lze jednoduše zakódovat ID jako čtyři bytové hodnoty. Dále bude následovat kódování samotného bloku, které bude složitější. Algoritmus kódování lze popsat následujícím způsobem:

1. Máme 63 celočíselných hodnot se znaménkem seřazených v „cik cak“ pořadí. Vezmeme první hodnotu.
2. Pokud je kódovaná hodnota nula a všechny následující kódované hodnoty jsou nulové, kódujeme jako hodnotu 40_{16} ($0100\ 0000_2$) a algoritmus končí.
3. Je-li kódovaná hodnota větší nebo rovna 40_{16} ($0100\ 0000_2$), tak se kóduje posledních 7 bitů čísla. Poslední bit je nastaven na hodnotu 1 což označuje číslo, které bude pokračovat. Provedeme rotaci kódované hodnoty o 7 bitů vpravo a pokračujeme znovu bodem 3.
4. Kódovaná hodnota je menší než 40_{16} ($0100\ 0000_2$). Proto ji můžeme kódovat na 6 bitů. Sedmý bit označuje znaménko (0 - kladné a 1 - záporné). Osmý (poslední) bit je nastaven na hodnotu nula, což značí konec kódované hodnoty.
5. Vybereme další hodnotu pro kódování a pokračujeme bodem 2.

Například máme zakódovat následující blok:

$$\begin{bmatrix} 272 & -126 & 12 & -3 & 0 & \cdots & 0 \\ 54 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 74 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 5 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 \end{bmatrix}$$

První hodnota se ukládá zvlášť, proto ji nebudeme dále uvažovat. Uvedený blok budeme procházet v „cik cak“ pořadí a získáme tak posloupnost:

$$-126, 54, 74, 0, 12, -3, 0, 0, 5, 0, 0, \ldots$$

Hodnota $126_{10} = 7E_{16} = 111\ 1110_2$. Jedná se o hodnotu větší než 40_{16}. Proto zakódujeme nejprve posledních 7 bitů jako $FE_{16} = 1111\ 1110_2$, kde první bit označuje, že bude pokračovat další část. Po rotaci o 7 bitů vpravo, které jsme zakódovali nám zůstane hodnota $0_{16} = 0_2$. Tuto hodnotu kódujeme přímo. Protože se jedná o záporné číslo bude mít sedmý bit hodnotu jedna a hodnotu kódujeme jako $40_{16} = 100\ 0000_2$. Zakódovali jsme celou hodnotu a pokračujeme k hodnotě následující.

Hodnota $54_{10} = 36_{16} = 11\ 0110_2$. Hodnotu lze zapsat přímo na 6 bitů, proto ji kódujeme přímo. Jedná se o kladné číslo, které je kódem přímo $36_{16} = 11\ 0110_2$. Pri zpracování hodnoty $74_{10} = 4A_{16} = 100\ 1010_2$ kódujeme nejprve prvních 7 bitů jako $CA_{16} = 1100\ 1010_2$ a následně zbytek $0_{16} = 0_2$ kódujeme přímo, protože se jedná o kladné číslo menší než 64. Následující hodnoty, nula a dvanáct, se kódují opět přímo bez jakýchkoli úprav, každá hodnota na jeden byt. Hodnota -3 se kóduje jako 3 s doplněným znaménkovým bitem na kód $43_{16} = 0100\ 0011_2$. Následující hodnoty se kódují přímo. První nula za číslem 5 bude kódována jako hodnota $40_{16} = 100\ 0000$ a tím kódování končí.

Výsledkem bude kód, který lze v šestnáctkové soustavě zapsat jako:

FE 40 36 CA 00 00 0C 43 00 00 05 40

Výsledný kód lze zapsat na 12 bytů. Za povšimnutí stojí, že druhá hodnota 40_{16} je shodná s ukončovacím znakem, který označuje nulové zakončení bloku. Tuto hodnotu odlišíme jednoduše tak, že se nejedná o začátek číselné hodnoty, proto předchozí byt začíná bitem s hodnotou nula. Ukončovací znak vždy předchází ukončení kódované hodnoty.

Dekódování v podstatě vyplývá z kódování a jedná se pouze o opačný postup. Při ukládání jsou bloky seřazeny vzestupně podle jejich ID a jsou ukládány všechny do jednoho souboru. Díky tomu nemusíme ukládat jejich ID, to je dáno pořadím bloku v souboru. Kvůli tomu současná implementace neumožňuje mazání bloků. Jak bylo uvedeno dříve, při ukládání se využívá aritmetické komprese.

Uložení zbytku hodnot

Pro každý soubor musíme následně uložit stejnosměrné (DC) složky hodnot a reference na bloky. Pro každou z uvedených částí je vytvořen jeden binární soubor. Hodnoty se ukládají jednoduše seřazené za sebou při použití adaptivního aritmetického kódování. Při ukládání se zapisují všechny složky (jasová i barevné) do jednoho souboru. Ukládání stejnosměrných složek navíc využívá výpočtu rozdílu sousedních hodnot, jak je popsáno v kapitole 2.1.8.

3.2.6 Výsledné velikosti souborů

Nyní se dostáváme k jedné z nejpodstatnějších kapitol. Cílem komprese je minimalizovat objem dat. Naneštěstí se popisovaná metoda komprese neukázala jako vhodná. Spíše naopak. Prvním testem bylo opět uložení používaných šesti obrazů samostatně do aplikace. Výsledný objem dat byl porovnán s uložením do formátu JPEG provedeného za pomoci knihovny technologie Java[14].

Při ukládání dat se ukládaly nejprve barevné složky, následně složka jasová. Tento postup napomáhá lepší kompresi při použití adaptivního aritmetického kódování. Nastavení podobnosti bylo stejné jako v předchozích případech. V barevných složkách hledáme naprostou shodu. U složky jasové se spokojíme s podobností 5 MSE. Predikce nul a sum jsou nastaveny na hodnotu 5 pro jasovou složku. Pro barevnou na hodnotu nula. Výsledky měření jsou shrnuty v tabulce 3.3.

Z výsledků je patrné, že především kódování bloků je v mém případě podstatně horší než v případě JPEG komprese. JPEG využívá navíc RLE kompresi, kterou kombinuje s kompresí Huffmanova kódování. Při pokusech přidat RLE kompresi do mé práce, jsem došel pouze k dalšímu zvětšení objemu dat. Hlavním důvodem je, že JPEG kombinuje RLE přímo s entropickým kódováním. V mém případě by bylo potřeba upravit kódování popisované v předchozí kapitole.

[14]Tento způsob byl zvolen především proto, abych měl jistotu, že použité kvantizační tabulky a hodnoty se shodují.

Obraz	Lena	Baboon	Fruits	Rocks	Lake	Cows
Velikost JPEG [KiB]	36,90	75,43	38,37	650,46	476,37	624,06
Velikost bloků [KiB]	27,25	89,23	27,05	439,27	238,21	484,66
Velikost DC složek [KiB]	12,38	12,79	11,73	160,70	114,23	123,31
Velikost odkazů [KiB]	18,41	26,68	16,10	229,00	152,97	185,23
Velikost celkem [KiB]	58,04	128,70	54,87	828,96	505,41	793,20

Tabulka 3.3: Tabulka měření objemu uložených dat.

Druhý problém při ukládání dat je, že uložených bloků může být velké množství a na referenci potřebujeme čtyřbytové hodnoty. I kdyby se nám podařilo ukládat bloky a stejnosměrné složky ve stejné velikosti jako JPEG komprese, bude výsledná velikost přibližně stejná jako velikost souboru při uložení JPEG. Jinými slovy velikost úspory dat na stejných nebo podobných blocích je přibližně stejná jako velikost, kterou zaberou navíc reference na bloky.

Jak bylo vidět již z předchozích výsledků, využití podobnosti je vhodné u méně zašuměných obrazů. U obrazů, kde existuje malé množství stejných nebo podobných bloků, dochází k nárůstu objemu dat.

Stejně jako v minulých kapitolách i nyní vyzkoušíme změnu velikosti dat při naplněné databází. Pro plnění bylo použito opět popisovaných 14 obrazů. Při vkládání bloků byla zanedbána podobnost a byly vkládány pouze unikátní bloky. Jako patnáctý byl vložen testovaný vzorek se stejnými pravidly podobnosti a ukládání jako v případě vkládání jednoho obrazu. Měřila se změna velikosti dat. Hodnoty stejnosměrných složek se nemění, a proto nejsou uvedeny ve výsledcích měření. Výsledky jsou zobrazeny v tabulce 3.4.

Obraz	Lena	Baboon	Fruits	Rocks	Lake	Cows
Velikost JPEG [KiB]	36,90	75,43	38,37	650,46	476,37	624,06
Velikost bloků [KiB]	27,02	89,11	26,84	438,94	238,03	484,46
Velikost odkazů [KiB]	25,06	30,53	23,21	323,18	231,46	258,85
Velikost celkem [KiB]	64,46	132,43	61,78	922,83	583,72	866,62

Tabulka 3.4: Tabulka měření objemu uložených dat při naplněné databázi.

Z výsledků je patrné, že ani větší množství obrazů v databázi nezlepší vlastnosti komprese. Spíše naopak. Velikost uložených bloků je o něco málo menší právě proto, že máme větší databázi pro vyhledávání. Ale rozdíl je velmi malý. Počet bloků, které již jsou v databázi a nemusíme je ukládat, je poměrně malý. Navíc nové bloky, které ukládáme, jsou většinou ve složce Y a dá se předpokládat, že budou poměrně složité, proto budou pro uložení potřebovat více prostoru nežli jednoduché bloky složek barevných.

Naopak nám velkou měrou vzrostla velikost dat pro uložení odkazů. To je způsobeno tím, že v jednom obrazu máme málo bloků a odkazy mají podobné a malé hodnoty. Proto je adaptivní aritmetická komprese mnohem účinnější.

3.2.7 Náročnost kódování

Popisovaná metoda komprese je asymetrická. Komprese je mnohem náročnější než dekomprese. Při kompresi je potřeba procházet velké množství bloků, které je potřeba porovnat. I s pomocí predikce a cache pro urychlení je množství prohledávaných bloků obrovské. Algoritmická složitost je $O(NM)$, kde N je počet vstupních bloků obrazu a M počet uložených bloků v databázi. Právě použitím cache můžeme algoritmus značně urychlit.

Při ukládání dat je potřeba držet všechny uložené bloky stále v paměti, protože jejich načítání a ukládání by velmi zdržovalo výpočet. Paměťovou náročnost lze uvažovat jako $O(M + N)$. Při velkém počtu obrazů (řádově desítky až stovky) by to mohl být problém.

Oproti tomu načítání obrazu je mnohem rychlejší, protože hledáme blok přímo podle jeho ID a víme, že uvedený blok existuje právě jeden. Při použití hash tabulek nebo stromů lze výpočetní složitost uvažovat jako $O(N \log_2 M)$. Všechny bloky máme uložené v paměti už kvůli kompresi, a proto stačí pro dekompresi udržovat pouze reference na stejné bloky.

Při implementaci nebyl kladen důraz na optimalizaci použitých metod. Šlo nám především o získání údajů o vhodnosti použití uvedené komprese. Existuje proto mnoho míst, které je možné razantním způsobem zrychlit. Existuje mnoho možností paralelního zpracování úlohy až k použití GPU pro výpočet.

3.2.8 Použití větších bloků

Další možnou alternativou na zlepšení vlastností algoritmů je použít větší bloky. Proto jsem testoval bloky o velikosti 16×16. Tím dojde ke zmenšení počtu bloků na čtvrtinu. Problém byl v tom, že jsem s těmito bloky provedl poměrně málo měření. Kvantizační hodnoty byly odvozeny od kvantizační matice 8×8. Při zachování stejných prahových hodnot podobností a stejných prediktivních metod bylo dosaženo lepších výsledků než při použití menších bloků. Otázkou zůstává, zda nebyly kvantizační hodnoty nastaveny příliš velké a zda nedošlo ke zhoršení kvality obrazů. Výsledné velikosti byly v některých případech větší a někdy menší než velikost při uložení do formátu JPEG. Obraz 3.10d byl větší o 8%, obraz 3.10e byl menší přibližně o 18% a obraz 3.10f byl přibližně stejně

velký jako v případě formátu JPEG. Bylo testováno pouze vložení prvního obrazu. Při vkládání více obrazů měla aplikace veliké paměťové nároky.

Z uvedených výsledků lze předpokládat, že při použití větších bloků bychom mohli dostat možná lepší výsledky, ale razantní zmenšení velikosti ukládaných dat (alespoň kolem 50%) pravděpodobně tímto způsobem nedosáhneme.

4 Závěr

Původní myšlenkou bylo vytvořit aplikaci, která se bude chovat podobně jako operační systém a bude stejně jednoduché zacházet s uloženými daty jako při práci s běžnými daty uloženými na pevném disku. Při předběžném zkoumání uvedeného řešení jsem přišel na to, že naplnění této myšlenky je velmi komplikované až nemožné. Místo aplikace, která bude simulovat souborový systém, jsem se rozhodl implementovat standardní desktop aplikaci, která bude co nejvíce uživatelsky příjemná. Domnívám se, že se toho podařilo dosáhnout hlavně díky použití „drag and drop“ a jednoduchých kontextových menu.

Implementovaná aplikace je jednoduchým nástrojem pro správu a strukturování dat. Navíc umožňuje doplnění o automatickou kompresi ukládaných dat. Způsob uložení nebo komprese je vhodným způsobem volen podle typu ukládaných dat. V současné podobě obsahuje aplikace tři různé způsoby uložení dat – původní nezměněný formát, uložení zástupce v případě, že soubor je již v aplikaci uložen, a použití komprese obrazu na základě podobnosti dat, kterou se zabývá tato práce. Aplikaci je možné jednoduchým způsobem rozšířit o nové kompresní algoritmy. Celý systém je stavěn co nejvíce komponentovým způsobem, a proto je možné i další rozšiřování celého zbytku aplikace. Není těžké přidat souborům parametry, které mohou některé nové kompresní algoritmy vyžadovat. Mělo by být možné doplnit aplikaci o další perspektivy vzhledu pro různé zobrazení. S touto myšlenkou byl systém navrhován, i když v tomto případě by byla úprava komplikovanější.

Zkoumaná kompresní metoda dopadla o poznání hůře. Celá metoda měla vycházet z JPEG komprese. Po převodu do YC_rC_b barevného prostoru, rozdělení na bloky, Fourierově transformaci a kvantizaci jsem chtěl vyhledat stejné nebo podobné bloky a ukládat pouze reference na ně. Nejprve jsem sice zjistil, že obrazy opravdu obsahují velké množství stejných nebo podobných bloků. Většina těchto bloků byla nalezena v barevných složkách. Při ukládání dat však dochází k tomu, že objem dat potřebný pro uložení referencí na bloky je přibližně stejný jako ušetřený objem získaný uložením pouze unikátních nebo podobných bloků. Při testování kvality se ukázalo, že pro zachování kvality obrazu je nutné brát v potaz bloky unikání nebo velmi podobné. Jakmile jsem použil bloky méně podobné, docházelo ke značné ztrátě kvality. Kvůli tomu podobnost bloků nepřinesla příliš velkou výhodu. V současné implementaci je kompresní poměr algoritmu dokonce podstatně horší než výchozí JPEG metoda. To je způsobeno především kódováním a ukládáním dat, které v mojí implementaci není rozhodně optimální. Určitě by bylo možné popisovanou metodu zlepšit. Domnívám se, že bychom dosáhli maximálně podobných výsledků jako JPEG komprese.

Druhým aspektem kompresní metody je její rychlost. Tato práce se rychlostí příliš nezabývá, protože jsem zkoumal především kvalitu komprese a zda-li je možné popisova-

nou myšlenku použít. Hlavním problémem s rychlostí je hledání bloků. Databáze bloků obsahuje již po několika uložených obrazech velké množství bloků a nalezení podobného bloku je poměrně náročná operace. Navíc většina výpočtů není optimalizovaná a rozhodně by bylo možné současnou implementaci vylepšit. Už jenom proto, že byly použity téměř výhradně knihovny implementované za pomoci technologie Java. Další možností urychlení je použití paralelního výpočtu nebo použití GPU.

Určitou možností pro zlepšení vlastností uvedené kompresní metody je použití větších bloků než 8×8. V tomto ohledu jsem zkusil pouze několik málo testů s bloky o velikosti 16×16. Výsledky ukazují jisté zlepšení kompresního poměru. Zlepšení však není nijak markantní.

Aby byla kompresní metoda použitelná, musí mít podle mého názoru kompresní poměr alespoň poloviční oproti metodě JPEG a přibližně stejnou nebo podobnou časovou náročnost. Z uvedených výsledků se domnívám, že i při všech možných vylepšeních této metody bychom se na tuto úroveň nedostali.

Literatura

[ŽBSF04] Jiří Žára, Bedřich Beneš, Jiří Sochor, Petr Felkel: *Moderní počítačová grafika.* Vydání první. Computer Press, 2004. ISBN 80-251-0454-0.

[Wró04] Piotr Wróblewski: *Algoritmy: Datové struktury a programovací techniky.* Překlad Marek Michalek a Bogdan Kiszka. Vydání první. Computer Press, 2004. ISBN 80-251-0343-9.

[AN07] Jim Arlow, Ila Neustadt: *UML 2 a unifikovaný proces vývoje aplikací: Objektově orientovaná analýza a návrh prakticky.* Překlad Bogdan Kiszka. Vydání první. Computer Press, 2007. ISBN 978-80-251-1503-9.

[ŠHB08] Milan Šonka, Václav Hlaváč, Roger Boyle: *Image processing, analysis, and machine vision.* Vydání třetí. Thomson, 2008. ISBN 978-0-495-08252-1.

[Umb05] Scott E. Umbaugh: *Computer imaging: digital image analysis and processing.* Vydání první. Taylor & Francis, 2005. ISBN 0-8493-2919-1.

[PS06] Lakshman Prasad, Alexei N. Skourikhine: Vectorized image segmentation via trixel agglomeration. *Pattern Recognition*, April 2006, Volume 39, Issue 4, Graph-based Representations, Pages 501-514. ISSN 0031-3203, DOI: 10.1016/j.patcog.2005.10.014.

[WBSS04] Z. Wang, A. C. Bovik, H. R. Sheikh and E. P. Simoncelli: Image quality assessment: From error visibility to structural similarity. *IEEE Transactions on Image Processing*, duben 2004, ročník 13, číslo 4, strany 600-612. Dostupné z `<http://www.cns.nyu.edu/pub/eero/wang03-reprint.pdf>`.

[W01] Přispěvatelé Wikipedie [online], *Wikipedie: Otevřená encyklopedie* Dostupná z `<http://cs.wikipedia.org/>` a `<http://en.wikipedia.org/>`.

[WE09] Přispěvatelé Wikipedie, Entropie [online], Wikipedie: Otevřená encyklopedie, c2009, Datum poslední revize 24. 12. 2009, 20:03 UTC, [citováno

28. 12. 2009]. Dostupné z <http://cs.wikipedia.org/w/index.php?title=Entropie&oldid=4743798>.

[WF10] Přispěvatelé Wikipedie, Fourierova transformace [online], *Wikipedie: Otevřená encyklopedie*, c2010, Datum poslední revize 16. 02. 2010, 08:53 UTC, [citováno 21. 02. 2010]. Dostupné z <http://cs.wikipedia.org/w/index.php?title=Fourierova_transformace&oldid=4968220>.

[WP10] Přispěvatelé Wikipedie, Povodí [online], *Wikipedie: Otevřená encyklopedie*, c2010, Datum poslední revize 15. 02. 2010, 15:50 UTC, [citováno 3. 03. 2010]. Dostupné z <http://cs.wikipedia.org/w/index.php?title=Povod%C3%AD&oldid=4965117>.

[WM10] Přispěvatelé Wikipedie, Model-view-controller [online], *Wikipedie: Otevřená encyklopedie*, c2010, Datum poslední revize 18. 02. 2010, 05:56 UTC, [citováno 6. 05. 2010]. Dostupné z <http://cs.wikipedia.org/w/index.php?title=Model-view-controller&oldid=4977235>.

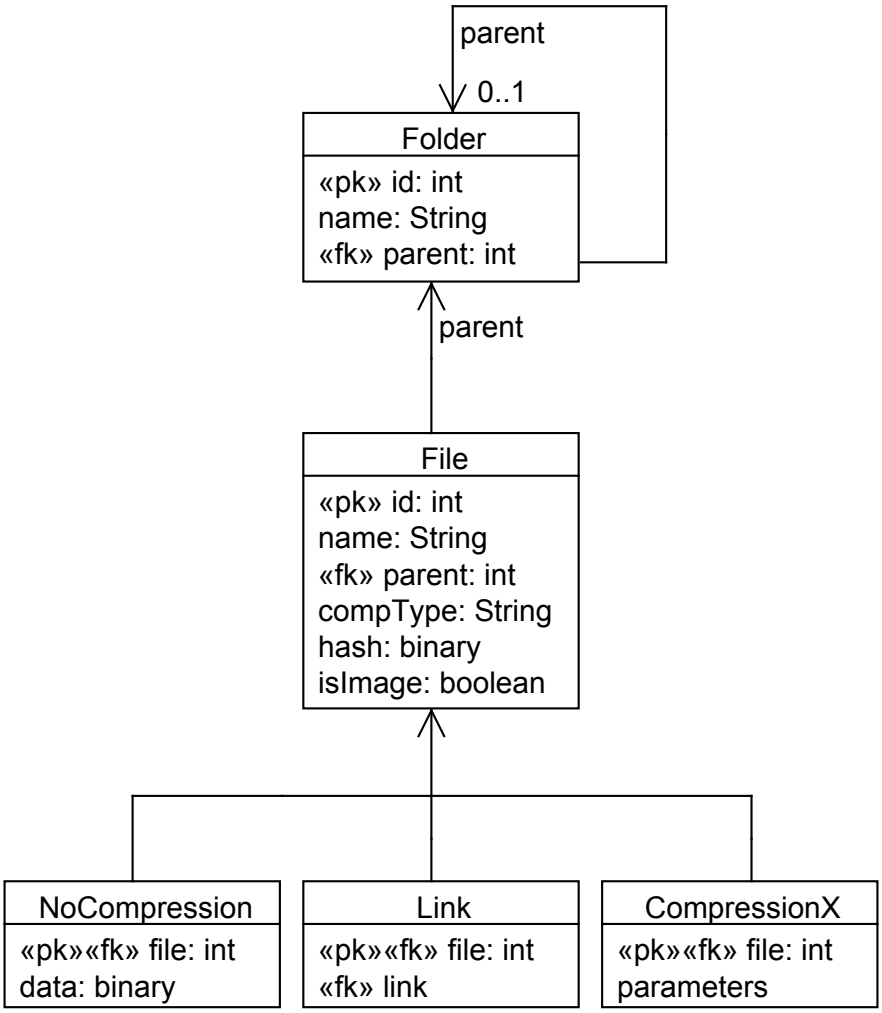
[Canny] *Canny edge detector*[online aplikace]. Poslední úprava 18.7.2008. University of Groningen: N. Petkov a M.B. Wieling. Dostupné z <http://matlabserver.cs.rug.nl/>.

[GIMP] *GIMP*[počítačový program]. Ver. 2.6.7. Spencer Kimball, Peter Mattis a vývojový tým GIMP. Dostupné z <http://www.gimp.cz/>.

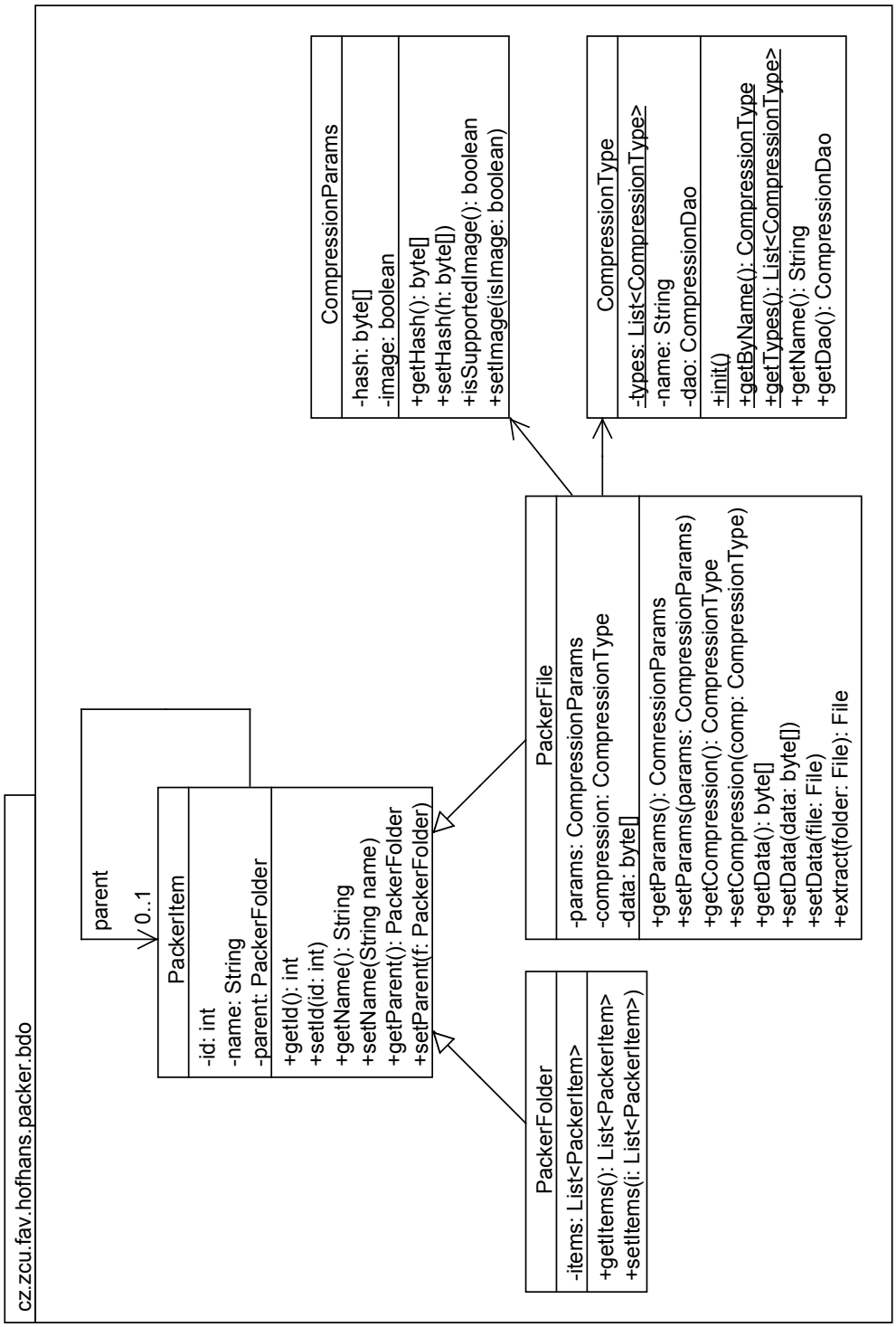
[HSQLDB] *HSQLDB*[SQL relační databáze]. Ver. 1.9.0-rc6. The HSQL Development Group. Dostupné z <http://hsqldb.org/>.

A Přílohy

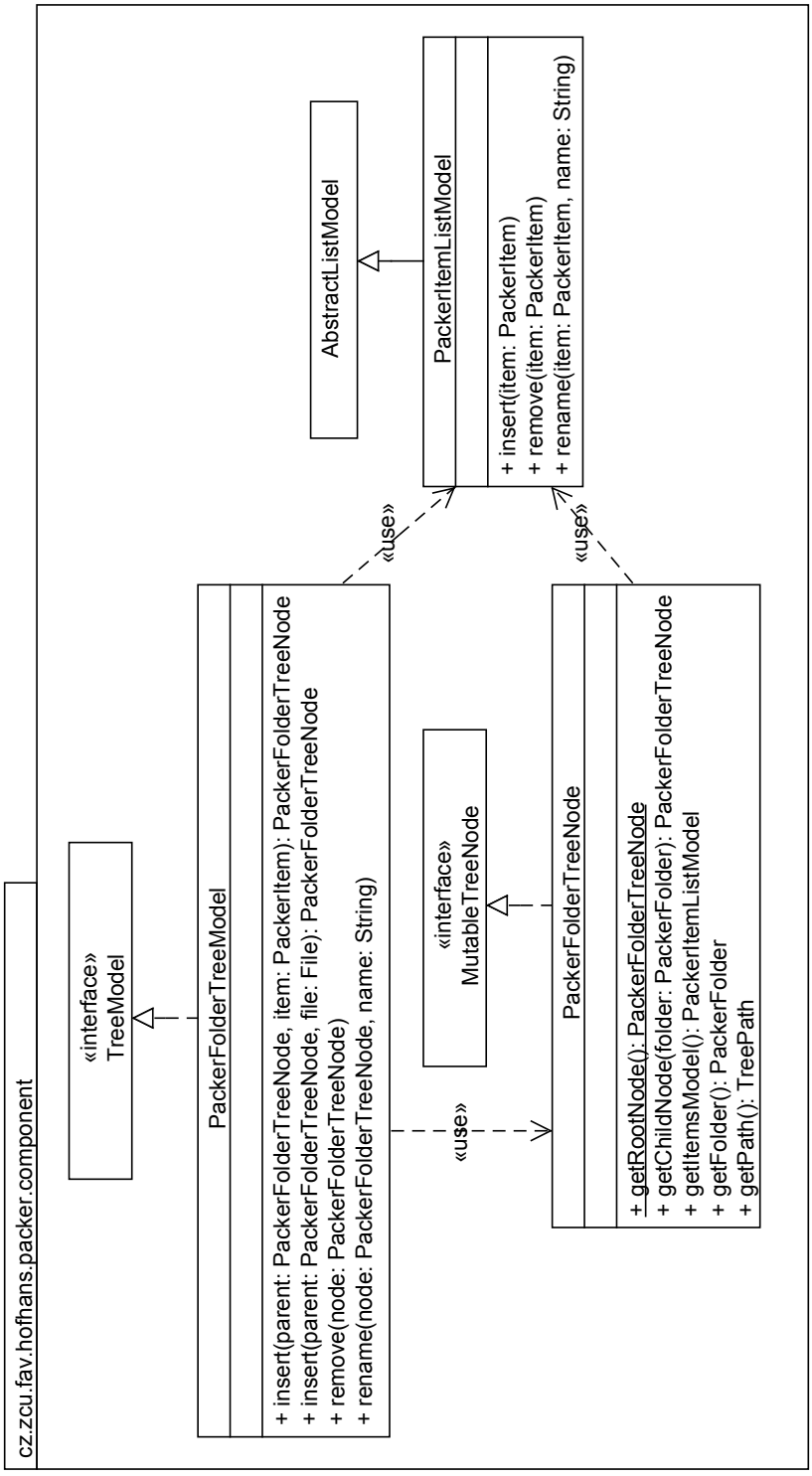
A.1 Diagramy



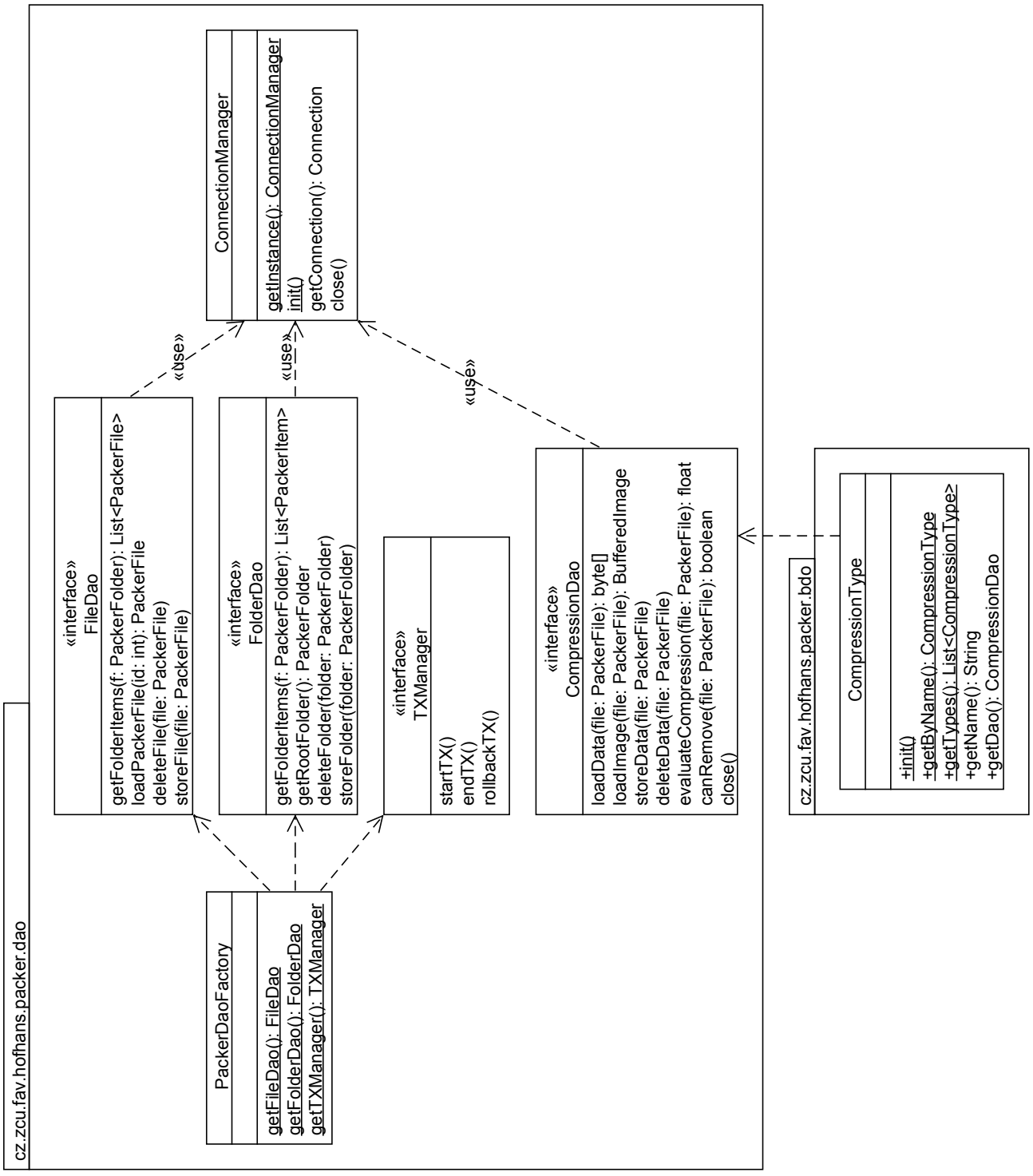
Obrázek A.1: Databázové schéma.



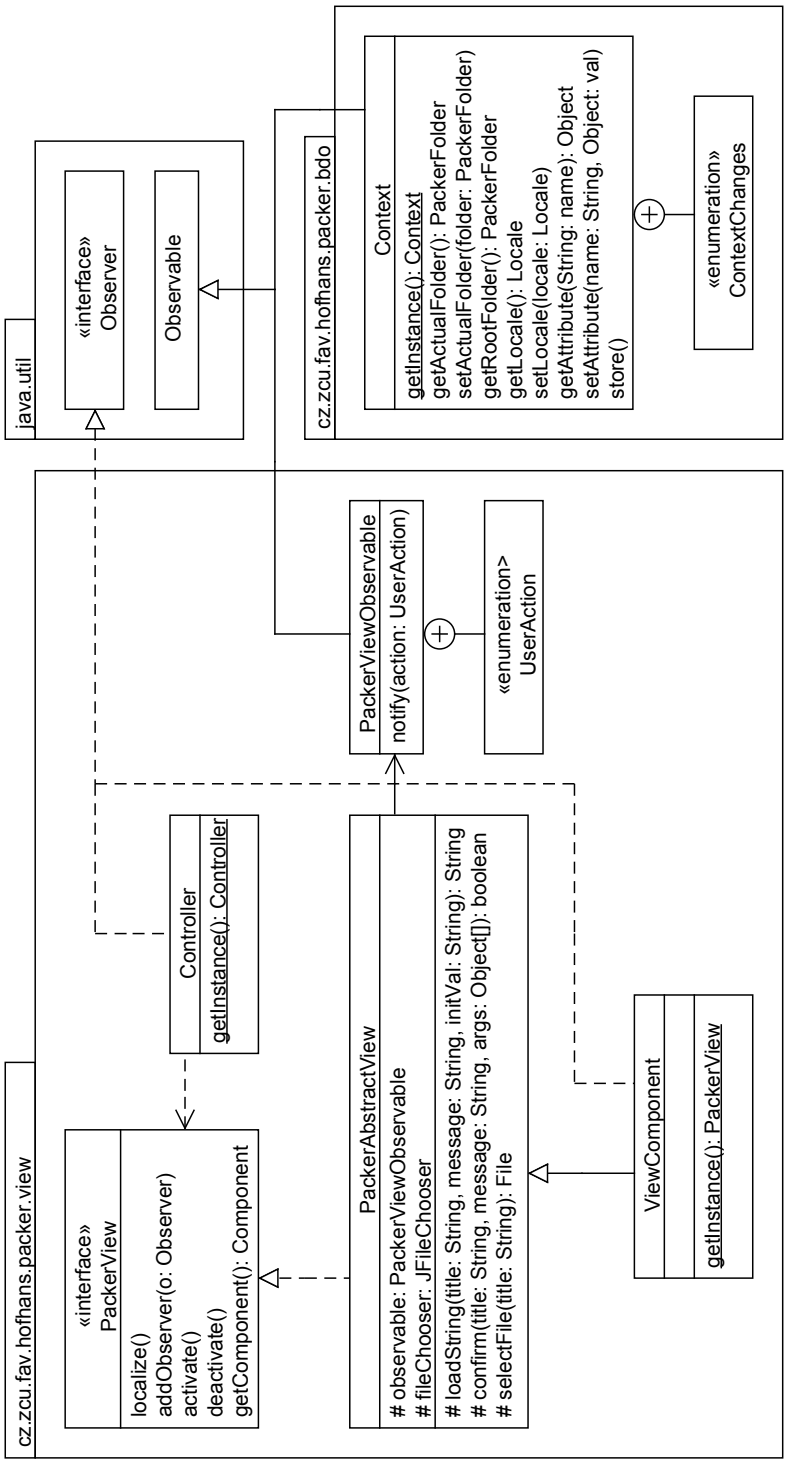
Obrázek A.2: Diagram datových tříd.



Obrázek A.3: Diagram modelu aplikace.



Obrázek A.4: Diagram rozhraní pro perzistenci dat.



Obrázek A.5: Model prezentační vrstvy.

A.2 Použité obrazy

Obrázek A.6: Lena.

Obrázek A.7: Baboon.

Obrázek A.8: Fruits.

Obrázek A.9: Rocks.

Obrázek A.10: Lake.

Obrázek A.11: Cows.

A.3 Použité knihovny

Při implementaci bylo použito několik externích knihoven. Všechny jsou implementovány za pomoci technologie Java. Zde je jejich soupis:

Compression via Arithmetic Coding – knihovna umožňující aritmetickou kompresi. Vydaná pod Apache/BSD licencí. Podrobnější informace na stránkách `http://www.colloquial.com/ArithmeticCoding/`.

HSQLDB – SQL relační databáze. Vydaná pod licencí odvozenou z BSD licence. Podrobnější informace na stránkách `http://hsqldb.org/`.

Java Advanced Imaging Image I/O Tools – knihovna pro komplexní práci s obrázky. Podrobnější informace včetně licence lze nalézt na stránkách `http://java.sun.com/products/java-media/jai/downloads/download-iio-1_0_01.html`.

JTransforms – knihovna umožňující rychlou Fourierovu transformaci včetně diskrétní kosinové transformace, vydávaná pod trojlicencí MPL/LGPL/GPL. Další informace viz `http://sites.google.com/site/piotrwendykier/software/jtransforms`.

JUnit – jednotkové testy pro technologii Java, licence CPL. Domovská stránka `http://www.junit.org/home`.

A.4 Obsah CD

- Data
  - Rozsireri – 14 obrazů používaných pro naplnění databáze daty
  - Zakladni – základních šest testovaných obrazů
- Dokumentace
  - JavaDoc – kompletní JavaDoc dokumentace aplikace
  - Zdroj_dokumentace – zdrojové soubory textu diplomové práce
  - diplomova_prace.pdf – text diplomové práce
- JRE
  - jre-6u20-windows-i586.exe – instalační balíček JRE - prerekvizita instalace aplikace
- Zdrojove_soubory – kompletní zdrovjové soubory aplikace
- HPacker.exe – instalační balíček
- HPacker.jar – instalační balíček ve formátu JAR
- HPacker.zip – přeložená spustitelná aplikace pouze zabalená do formátu ZIP

A.5 Uživatelská dokumentace

A.5.1 Instalace a spuštění

Před samotnou instalací je potřeba nainstalovat JRE (Java Runtime Environment). Aplikace byla testována na verze 1.6, ale měla by fungovat i ve verzích 1.5 nebo vyšších. Pro instalaci aplikace byl vytvořen standardní instalátor, který stačí spustit a postupovat podle kroků průvodce. Před spuštěním aplikace je potřeba nainstalovat prázdnou databázi nebo provést import již vytvořené databáze.

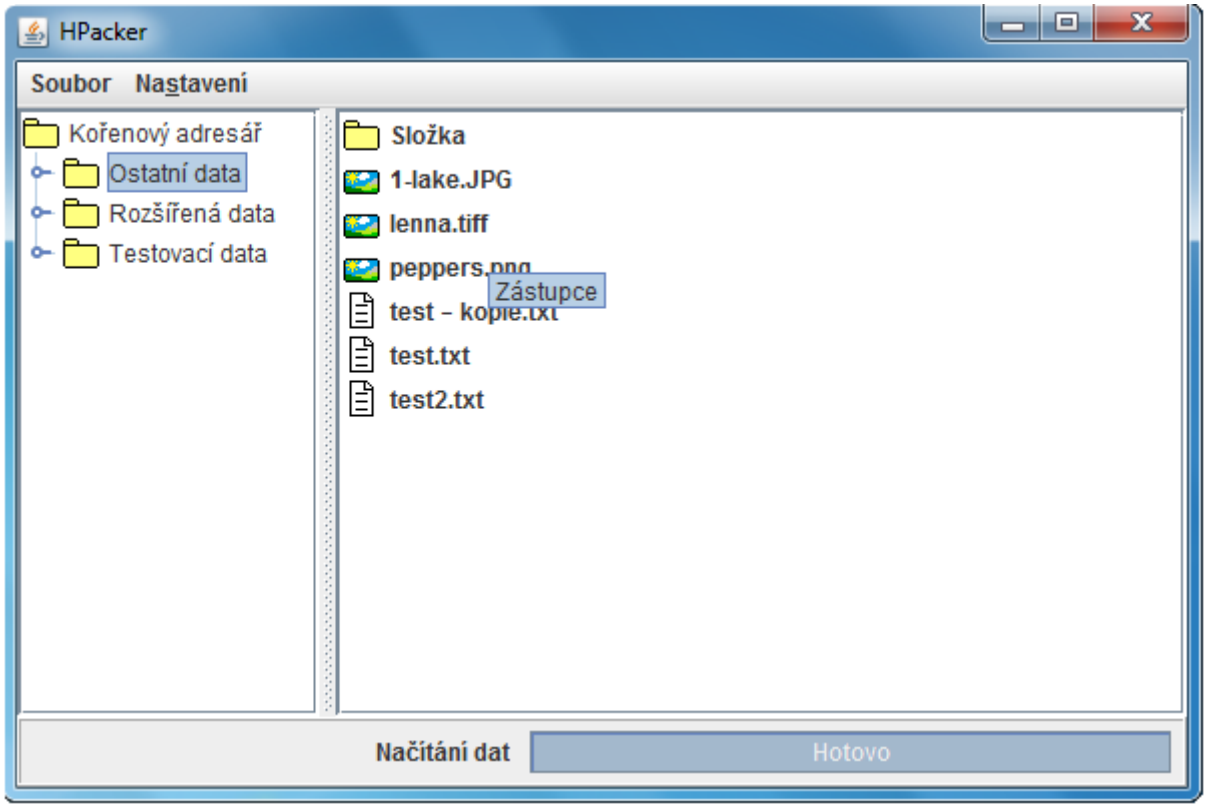
Pro prostředí operačních systémů Windows byly vytvořeny `BAT` soubory pro spouštění aplikace a pomocných nástrojů. Po instalaci je možné spustit aplikace souborem `run.bat`. Aplikace umožňuje vytvoření zálohy dat a jejich načtení. Vytvořit zálohu je možné pomocí `makeBackup.bat`. Následně je možné ji načíst za pomoci `loadBackup.bat`. Při načtení databáze přijde uživatel o všechna stávající data. Pro vymazání stávajících dat a instalace nové prázdné databáze slouží soubor `install.bat`.

Při použití více parametrů nebo pro spouštění v jiných operačních systémech je nutné aplikace spouštět z konzole. Základní možnost spuštění je:
`java -jar packer.jar [install|backup]`
Bez parametrů se jedná o spuštění aplikace. Parametr `install` udává, že budou smazána stávající data a vytvořena nová čistá databáze.

Druhou možností je argument `backup`. Po tomto argumentu musí následovat další parametry:
`java -jar packer.jar backup [-i|-e] [file]`
Parametry `-i` a `-e` rozhodují o tom, zda se bude jednat o import nebo export. Volitelně je možné zadat jméno souboru, který se bude importovat nebo exportovat. Pokud soubor není uveden, zobrazí se průvodce pro přidání souboru.

A.5.2 Ovládání

Ovládání aplikace je velmi jednoduché a intuitivní. Základní možnosti ovládání jsou umístěny do menu aplikace. Ostatní volby lze zobrazit pomocí kontextového menu kliknutím pravého tlačítka na prvek, jehož volby chceme zobrazit. Klikneme-li například pravým tlačítkem na adresář, zobrazí se volby pro práci s uvedeným adresářem. Ukázka aplikace je na obrázku A.12.



Obrázek A.12: Ukázka aplikace.

Přidávat data do aplikace je možné pomocí kontextových menu nebo použitím „drag and drop“. To znamená, že je možné soubory nebo adresáře přetáhnout myší do aplikace. Stejným způsobem se provádí vytažení souborů z aplikace. Pokud chceme přetáhnout položku (soubor nebo adresář) ven z aplikace, je vhodné to činit po jednom souboru. Dekomprese více souborů může být totiž náročnější a trvat delší dobu. Při přetahování je nutné vyčkat na kompletní dekompresi přetahovaného obsahu. Dekomprese je kompletní, když vidíme u kurzoru myši symbol přesunu nebo kopírování. Přesun z aplikace není podporován a provádí se vždy jen kopírování.

Aplikace obsahuje stavový řádek s progress barem, který zobrazuje průběh prováděné operace. Zobrazuje například načítání dat při startu aplikace nebo průběh komprese. Zobrazuje pouze průběh jedné operace. Pokud přidáváme více souborů najednou, zobrazuje postupně průběh komprese jednotlivých souborů.

Při používání aplikace je možné zjistit způsob uložení jednotlivých souborů pomocí tooltipu. Najetím kurzoru myši nad uložný soubor se zobrazí způsob jeho uložení.

A.6 Možnosti rozšíření

Tato kapitola popisuje možnosti rozšíření implementované aplikace. Především se zabývá přidáním nové kompresní metody.

A.6.1 Přidání kompresní metody

Nejdůležitějším krokem je implementace rozhraní `CompressionDao` zobrazeného v diagramu A.4. Je nutné implementovat většinu zadaných metod. Význam jednotlivých metod je následující:

`byte[] loadData(PackerFile file)` – vrací data ve standardním formátu. Formát je volen podle názvu souboru, jak je v aplikace uložen. Pokud se jedná např. o „soubor.png“, lze data přímo uložit do souborového systému jako soubor.png.

`BufferedImage loadImage(PackerFile file)` – tuto metodu lze použít jen v případě, že se jedná o podporovaný obrázek[1]. Pokud ano, metoda vrací standardní obrazovou abstrakci technologie Java.

`storeData(PackerFile file)` – uložení dat souboru do systému. Používá se pouze při prvním uložení, protože aplikace nepodporuje změnu dat. Metoda se nestará o uložení souboru do struktury dat, pouze o uložení jeho datové informace.

`deleteData(PackerFile file)` – trvalé odstranění dat, pokud je to možné.

`boolean canRemove(PackerFile file)` – test, zda je možné odstranit uvedený soubor. Pokud na tento soubor existuje nějaká vazba nebo nelze z jiného důvodu odstranit, vrací metoda `false` hodnotu.

`float evaluateCompression(PackerFile file)` – ohodnocení vhodnosti komprese. Metoda by měla otestovat vhodnost komprese a vrátit hodnotu z intervalu [0, 1]. Nula označuje nevhodnost komprese. Naopak hodnota jedna naznačuje optimální uložení. Speciální hodnota -1 označuje, že pro daný soubor není možné kompresi použít. V případě shodnosti ohodnocení u dvou různých kompresí se použije libovolná z nich. Uložení bez komprese vrací hodnotu nula. Uložení za pomoci zástupce vrací hodnotu 1 nebo -1 podle toho, zda byl nalezen stejný uložený soubor.

[1] Pokud se nejedná o podporovaný obrázek je možné vyhazovat výjimku (např. `PackerRuntimeException`).

`close()` – ukončení práce s kompresí. Předpokládá se, že komprese může používat vlastní data uložená v paměti (např. cache), které bude při ukončení aplikace potřeba uložit. Tato metoda slouží k finálnímu ukončení práce a uložení dat.

Většina popisovaných metod může vyhazovat chytatelnou výjimku `ValidationException`, která označuje problém při zpracování. Výjimka je chytaná a v lokalizované podobě zobrazena uživateli. Navíc je nutné implementovat tovární metodu třídy `getInstance()`, která vrací instanci kompresní metody. Tato metoda se volá za pomoci reflexe místo použití konstruktoru. Pro zařazení implementované třídy do aplikace je potřeba upravit konfigurační soubor `config\compression.properties`. Tento soubor obsahuje dvě hodnoty pro každý typ komprese: název komprese a cestu k implementované třídě. Typický příklad:

```
NEW_COMPRESSION.name=NEW_COMPRESSION
NEW_COMPRESSION.dao=cz.zcu.fav.hofhans.packer.dao.internal.NewCompressionDao
```

Položka `name` označuje název komprese, který je následně lokalizován. Druhý záznam s klíčem `name.dao` udává přesnou cestu, kde leží implementace kompresního rozhraní.

Konfigurace

Veškerá konfigurace aplikace se obsažena v adresáři `config`. Všechny konfigurační soubory jsou uloženy jako properties soubory. Konfigurační adresář obsahuje tři povinné konfigurační soubory a dva soubory, které jsou volitelné:

`configuration` – základní konfigurace aplikace. Obsahuje například výchozí velikost okna aplikace, cestu k adresáři dočasných souborů a výčet podporovaných lokalizací.

`connectionManager` – informace pro připojení k databázi. Konfiguruje se zde třída JDBC ovladače, URL databáze a přihlašovací údaje.

`compression` – slouží k definici používaných kompresních metod. Tento soubor byl popisován v předchozí kapitole.

`context` – slouží k uložení kontextu aplikace. Není určen k přímé editaci uživatelem nebo administrátorem systému.

`log` – volitelná konfigurace logování aplikace. Aplikace loguje za pomoci standardních logovacích tříd (nevyužívá se žádná externí knihovna).

Lokalizace

Lokalizace je implementována pomocí standardních tříd technologie Java. Výčet použitých lokalizací je uveden v základním konfiguračním souboru aplikace (viz předchozí kapitola). Pro konfiguraci se využívá kódů jazyků a oblastí definovaných standardy ISO (podrobněji viz JavaDoc API nebo příslušné standardy). Výchozí lokalizace aplikace je česká a anglická. Tomu odpovídají kódy 'cs' a 'en'. Samotné lokalizační soubory jsou uloženy opět v properties souborech. Každý soubor obsahuje jednu lokalizaci. Název souboru je potom `zdroj_lokalizace.properties`. Český zdroj je potom například `zdroj_cs.properties`. Pro podrobnější informace viz třídy `Locale` a `ResourceBundle` obsažené v API technologie Java.

Lokalizace názvů kompresních metod je uvedena v properties souborech s názvem `CompressionNames`, kde klíčem je název komprese uvedený v konfiguraci komprese a hodnotou je její lokalizace.

Třída `ValidationException` obsahuje vnitřní výčtový typ s výběrem možných validačních problémů. V případě potřeby je možné výčet rozšířit. Lokalizace jednotlivých validačních výjimek je uložena v properties souborech `ValidationException`.

Pro lokalizaci je potřeba dodat, že všechny properties soubory s lokalizací jsou uloženy v balíku `cz.zcu.fav.hofhans.resources`. Z definice properties souborů je nutné, aby byly uloženy v kódování ISO-8859-1. Je však možné vytvořit soubory v kódování UTF-8 a následně je transformovat (například za pomoci nástroje ANT). Takto vytvořené zdrojové soubory jsou uloženy v adresáři `src\resources\utf`.