Kódování a komprese dat 2022/2023
Projekt

Vojtěch Šíma
xsimav01@vutbr.cz

5. května 2023

1 Rozbor řešeného problému

V tomto projektu je cílem komprimovat obrázek ve formátu .raw, který je na vstupu algoritmu tak, aby došlo k co největší úspoře místa v paměti. S využitím přepínače -c dojde ke kompresi obrázku, jehož název je zadán s přepínačem -i. Tento komprimovaný soubor je uložen za využití kanonického Huffmanova kódu do souboru, který je zadán přepínačem -o. Cílem je co nejvíce zmenšit velikost souboru ale také zachovat všechny jeho vlastnosti, aby potom mohl být zpětně dekódován do původní podoby, k čemuž je využito přepínače -d a soubory jsou prohozeny. Na výstupu tak je původní obrázek ve formátu .raw.

2 Překlad a spuštění

K projektu je přiložen **Makefile**, který pomocí příkazu *make* vytvoří spustitelný soubor, který pak je spouštěn z příkazového řádku. Pomocí ./huff_codec -h je vyvolána nápověda pro bližší přiblížení využitých přepínačů.

3 Vlastní řešení

Řešení se nachází rozčleněno do pěti hlavní souborů, kdy v souboru **huff_codec.cpp** se nachází ošetření argumentů a v závislosti na zadaných přepínačích dochází k spuštění příslušného souboru pro kódování či dekódování.

3.1 Kódování

Pro kódování celého obrázku slouží soubor **code.cpp**. Jsou načteny všechny hodnoty z vstupního obrázku. V případě, že byl zadán přepínač -m, jsou tyto hodnoty upraveny tak, že jsou místo jednotlivých hodnot uloženy pouze rozdíly vzhledem k hodnotě předchozí (vyjma hodnoty první, která zůstává stejná). Na základě těchto hodnot jsou pak vypočítány jednotlivé četnosti výskytu znaků. Ty jsou upraveny tak, aby byly zachovány pouze nenulové četnosti znaků.

Dle doporučení z přednášky bylo využito hromady, do jejíž druhé poloviny jsou seřazeny od nejmenšího dříve zjištěné četnosti. Pak dojde k volání funkce *heapwork*, která podle velikosti hromady (toho, kolik četností různých znaků máme) provede rekurzivně daný počet běhů. Na konci jsou pak prvky v hromadě uloženy tak, že je jeden root a všechny ostatní prvky na něj přes jiné odkazují. Tato funkce způsobuje problém z časového hlediska které jsou popsány v 4 později.

Poté je volána funkce *vypocetdelek*, která na hromadě vypočte z druhé poloviny délky k rootu pomocí algoritmu Hirschberg – Sieminski.

Tyto délky pak jsou seřazeny společně se svými indexy (indexy zde jsou synonymem pro samotné hodnoty 0-255) ve funkci *bubble_sort*. Bylo potřeba zachovat stejné seřazení obou polí podle položek v poli s délkami. Po tomto seřazení již může být na základě délek od nejkratších sestavován samotný kanonický Huffmanův kód.

Následně jsou zjištěny délky jednotlivých kódů, aby mohly být správně uloženy. Do souboru jsou pak uloženy informace o velikosti pole s délkami jednotlivých znaků (1B), samotné pole kdy index značí délku a hodnota počet znaků zakódovaných s touto délkou. Součtem těchto hodnot je pak celkový počet znaků, kdy jsou jednotlivé znaky uloženy. Za touto hlavičkou jsou již ukládány jednotlivé zakódované znaky.

Vždy je přes indexy načteno o jaký znak se jedná, jaký má kód a jaké je délka kódu. Na základě této délky dochází k potřebnému počtu bitových posunů a vždy když dojde k načtení jednoho bajtu, dojde k zápisu do souboru. Na konci je ještě uložen poslední byte se zbytkem informace a za ním ještě jeden, poslední, ve kterém je informace o tom, kolik užitečných bitů se vyskytuje v předchozím, předposledním bajtu.

3.2 Dekódování

Pro kódování celého obrázku slouží soubor **decode.cpp**. Při dekódování je zjištěna délka souboru a je načtena hlavička s uloženými hodnotami. Dle přednášky jsou pak sestrojeny struktury (pole) **firstCode** (uchovává hodnotu prvního kódovaného slova pro všechny délky) a **firstSymbol** (určuje index prvního slova), které jsou využity pro dekódování kanonického Huffmanova kódu.

Jsou načítány jednotlivé bajty, které jsou procházeny dle algoritmu a pomocných struktur. Vždy když dojde ke zjištění, že již byl načten nějaký znak, dojde k zapsání do souboru. V případě využití přepínače -m je uchovávána předchozí hodnota tak, aby mohl být daný pixel vždy dopočítán. V předposledním bajtu je dle hodnoty bajtu posledního přečten pouze užitečný počet bitů.

3.3 Adaptivní kódování a dekódování

Adaptivní kódování je realizováno v souboru **code_adapt.cpp**, kdy dojde k načtení souboru do bufferu a pak dojde k rozdělení do jednotlivých bloků o velikost 8x8, které jsou postupně poslány do funkce *Code_block*.

V této funkci dochází k podobnému kódování jako při statické verzi. Díky problémům, které byly řešeny se nakonec nepodařilo realizovat samotné adaptivní kódování, které by vybíralo mezi horizontálním a vertikálním řešením a tudíž je prováděno pouze horizontální. Řešení je také omezeno na velikosti referenčních obrázků. Všechny operace jsou pro kódování prováděny podobně jako v **code.cpp**, navíc je však na první pozici v hlavičce uložena informace o tom, kolik bude načítáno celých bajtů s zakódovanými znaky. Tato hodnota je získána že jsou dopředu procházeny všechny znaky a zjištěna délka jejich kódů. Toto je nutné z hlediska toho, aby byl běh kódovací a dekódovací funkce ukončen ve správný čas pro daný blok.

Dekódování probíhá tak, že jsou procházeny jednotlivé bloky, kdy je vždy načtena hlavička, potřebný počet bajtů a vše je dekódováno do pole block. To je pak procházeno a jednotlivé hodnoty jsou vkládány na správné indexy v celkovém poli buffer, které po projití všech bloků uloženo do výsledného .raw souboru jako celek.

4 Problémy mého řešení

Mé řešení se potýká s určitými problémy. Jedním z hlavních problémů je poměrně velká časová náročnost, který je způsobena s tím, jak je při kódování pracováno s algoritmem, který dělá operace na hromadě. Využívám zde pomocná pole pro ignorování indexů a minim, což v kombinaci s opakovaným rekurzivním voláním a řazením na hromadě způsobuje to, že celé řešení v některých případech trvá opravdu velmi dlouho ve statické verzi a ještě více v adaptivní, kdy dochází k sestavování hromady opakovaně pro každý blok samostatně. Řešení však z hlediska počtu bitů na znak má dle mého dobré výsledky, což jsem vyhodnotil jako podstatnější než časovou složitost, která je však opravdu veliká.

Dalším problémem je problém při adaptivním dekódování s parametrem -m, na některých souborech i bez něj. Nepodařilo si mi ani po dlouhé době přijít na to, kde je chyba do řešení vpravena. Problém vychází z toho, že v průběhu čtení nedojde k správnému přečtení zakódovaných délek, dojde k posunutí o jeden či více bajtů a pro další blok již jsou tedy hodnoty posunuty, což vede na Segmentation fault.

Bohužel se toto řešení nepodařilo plně opravit a zprovoznit, což vedlo pak k tomu že nebylo realizováno adaptivní rozhodování o směru kódování jak již bylo zmíněno v 3.3 a výsledné adaptivní dekódování tak funguje pouze pro určité soubory.

Mezi soubory se vyskytuje nepravidelnost ve využívání statických a dynamických polí, která byla způsobená tím, že při hledání problémů došlo k různým změnám při snaze odhalit chybu a pak již nezbyl čas na výsledné sjednocení do jednoho stylu.

5 Vyhodnocení efektivity

Výpočet jednotlivých efektivit probíhal na školním serveru Merlin. Z hlediska efektivity komprimace se jedná o očekávané a zdařilé výsledky, z hlediska časové efektivity je sitauce horší a byla zmíněna dříve v 4.

SOUBOR	ENTROPIE	STAT	STAT -m	ADAPT	ADAPT -m
df1h.raw	8.00	8.00	1.00	4.87	2.25
df1hvx.raw	4.41	4.58	1.83	4.87	2.10
df1v.raw	8.00	8.00	1.00	4.87	2.23
hd01.raw	3.83	3.88	3.40	5.57	5.58
hd02.raw	3.64	3.70	3.33	5.37	5.50
hd07.raw	5.58	5.61	3.85	6.43	5.75
hd08.raw	4.21	4.23	3.52	5.52	5.55
hd09.raw	6.62	6.66	4.68	8.18	7.34
hd12.raw	6.17	6.20	4.39	7.30	6.50
nk01.raw	6.47	6.50	6.06	10.31	11.02
PRŮMĚR	5.70	5.736	3.306	6.329	5.382

Tabulka 1: Efektivita komprimace a entropie jednotlivých souborů. Tabulka zobrazuje počet bitů potřebný na zakódování jednoho pixelu

SOUBOR	STAT	STAT -m	ADAPT	ADAPT -m
df1h.raw	2.811 s	0.013 s	3.938 s	3.890 s
df1hvx.raw	0.051 s	0.338 s	4.146 s	4.307 s
df1v.raw	2.835 s	0.013 s	4.325 s	4.199 s
hd01.raw	3.003 s	2.909 s	8.521 s	7.798 s
hd02.raw	3.146 s	2.816 s	8.417 s	8.040 s
hd07.raw	2.923 s	1.148 s	7.044 s	5.499 s
hd08.raw	0.087 s	0.680 s	6.427 s	6.557 s
hd09.raw	2.678 s	2.564 s	8.863 s	6.665 s
hd12.raw	3.198 s	1.589 s	9.120 s	6.747 s
nk01.raw	2.335 s	2.863 s	10.318 s	11.200 s
PRŮMĚR	2.3067 s	1.4933 s	7.1119 s	6.4902 s

Tabulka 2: Časová efektivita komprimace jednotlivých souborů

6 Literatura

Přednášky k předmětu Kódování a komprese dat
https://moodle.vut.cz/pluginfile.php/434070/course/section/57982/lectures/KKO-03.pdf
https://moodle.vut.cz/pluginfile.php/434070/course/section/57982/KKO-04.pdf