KKO projekt: Kompresia obrazových dát s využitím statického a adaptivneho Huffmanovho kódovania
Meno a priezvisko: Dávid Bolvanský
Login: xbolva00

1 Úvod

Cieľom tohto projektu je vytvoriť program pre kompresiu šedotónových obrazových dát, kde sa uplatnia princípy statického a adaptívneho Huffmanovho kódovania.

2 Rozbor problému

V tejto kapitole sú predstavené princípy stojace za statickým a adaptívnym Huffmanovým kódovaním.

2.1 Statické Huffmanovo kódovanie

Na začiatku si algoritmus spočíta výskyty (frekvencie, váhy) znakov z abecedy vo vstupných dátach. Pokiaľ teda nie sú známe pravdepodobnosti výskytov znakov dopredu, Huffmanovo kódovanie sa stáva dvojpriechodovou metódou, kde v prvej fáze sa zbierajú počty výskytov symbolov a v druhej prebieha samotná tvorba kódových slov. Na základe týchto váh je následne zostavený binárny strom tak, že listové uzly reprezentujú samotné písmená z abecedy a hrany reprezentujú hodnotu 0 alebo 1. Platí, že častejšie vyskytujúce sa znaky sú umiestnené bližšie ku koreňu stromu. Hrany na ceste od koreňa k listom stromu tvoria jednotlivé kódová slova. Výsledný binárny kód je prefixový. Po získaní kódového slova pre každý znak je následne možné vykonať komprimáciu dát. Pre každý bajt vstupných dát sa na výstupe vypíše preňho odpovedajúce kódové slovo (sekvencia bitov).

Aby bola možná dekomprimácia dát, je nutné do výstupného súboru taktiež uložiť informáciu pre priradenie znaku k určitej sekvencii bitov. Možností je viacero, do hlavičky súboru sa môže uložiť celý binárny strom, prípadne znaky spolu s ich váhami, a pod.

Pre efektívnu reprezentáciu hlavičky sa používa kanonický Huffmanov kód. Algoritmus pre výpočet kanonického Huffmanovho kódu prevádza kódové slová z „klasického“ Huffmanového kódovania do kanonickej podoby. Následne v hlavičke stačí zachytiť už len informáciu o počte dĺžok kódových slov, počte znakov zakódovaných určitou dĺžkou kódového slova a znaky vstupnej abecedy (tj. znaky, ktoré sa vyskytovali vo vstupe).

2.2 Adaptívne Huffmanovo kódovanie

Adaptívne (dynamické) Huffmanovo kódovanie je jednopriechodová metóda, ktorá nepotrebuje poznať váhy znakov dopredu. Váhy znakov sa určujú a aktualizujú priebežne tak, ako sa načítavajú znaky zo vstupu. Metóda je výhodná v prípade, že lokálne váhy v dátach vykazujú odlišné charakteristiky voči globálnym váhami. Štruktúra kóderu a dekóderu je zrkadlová. V princípe sa na začiatku inicializuje strom, a následne sa čítajú znaky zo vstupu. Po zakódovaní či dekódovaní znaku je nutné aktualizovať Huffmanov strom. Je niekoľko možností ako aktualizovať strom, no medzi najvhodnejšie a najefektívnejšie možnosti patria algoritmy FGK / V. Algoritmus V vylepšuje algoritmus FGK a dosahuje vyššiu účinnosť kompresie.

2.3 Algoritmus FGK

Gallager dokázal, že binárny prefixový kód je Huffmanový kódom práve vtedy, ak má odpovedajúci kódovací strom súrodeneckú vlasnosť. Binárny strom má súrodeneckú vlastnosť, ak každý uzol okrem koreňa má súrodenca a pokiaľ je možné uzly zoradiť do monotónnej postupnosti podľa ich váh tak, že každý uzol má v postupnosti za suseda práve svojho súrodenca. Algoritmus FGK funguje na princípe vkladania znakov do stromu. V prípade, že sa zistí porušenie súrodeneckej vlastnosti, dochádza k transformácii stromu tak, aby súrodenecká vlasnosť platila. Existujú dva prístupy na inicializáciu stromu, kde strom na začiatku obsahuje všetky znaky so svojimi váhami. Druhý prístup inicializuje strom jediným uzlom (špeciálny NYT („Not Yet Transferred“) uzol) a následne po načítaní znaku, ktorý ešte nebol zakódovaný, sa na výstup vypíše kód NYT uzla. Následne sa NYT uzol rozdelí na nový NYT uzol a na nový listový uzol, ktorý bude obsahovať načítaný znak.

3 Popis implementácie

Program obsahujúci statické, adaptívne Huffmanove kódovanie a model je napísaný v jazyku C++11. Preklad prebieha s najvyššími optimalizáciami spolu s LTO pre zaručenie maximálneho výkonu výsledného programu. Ako v implementácii statického, tak aj v implementácii adaptívneho Huffmanovho kódovania sa vyskytovali

problémy, ktoré vyžadovali univerzálne riešenie použiteľné v oboch implementáciách. Pre ukladanie bitov do vektora bajtov bola vytvorená trieda `bitpacker`. Implementácia tejto triedy sa nachádza v súbore `bitpacker.cpp`. Trieda abstrahuje prácu nad ukladaním bitov do bajtov a ponúka metódy na pridávanie bitov do žiadanej podoby - pole bajtov. Následne pre prácu s uzlov v strome bola navrhnutý trieda `tree_node` s implementáciou v súbore `tree_node.cpp`. Uzol obsahuje informácie o znaku, ktorý obsahuje, o typu uzla, jeho váhe a poradí v strome. Obsahuje taktiež ukazovatele na svoje deti (synov) v strome a ukazovateľ na svojho rodiča. Metódy v tejto triede slúžia na prístup k vlastnostiam uzla a na ich zmenu. V C++ je možné zmeniť sémantiku operátorov a táto možnosť je v triede `tree_node` využitá a operátory porovnania dvoch uzlov porovnávajú uzly podľa ich váh. Kódové slovo je ukladané do vektora bitov, v C++ bol použitý špecializovaný kontajner `std::vector<bool>`.

4 Statické Huffmanove kódovanie

Implementácia je zapuzdrená v triede `static_huffman`, ktorá je v súbore `static_huffman.cpp`. V konštruktore triedy sa získajú počty výskytov (váhy) znakov abecedy vo vstupných dátach. Pre uloženie hodnoty váh (frekvencií) sa používa dátový typ `size_t`, kde jeho maximálne číslo je 18446744073709551615, a preto v implementácii sa neuvažuje nad pretečením tohto počítadla. Ak by sa mal tento nie moc reálny problém riešiť, všetky váhy by sa podelili dvomi a následne by sa v prípade potreby preusporiadal strom. V metóde `build` sa pre každý znak s nenulovou váhou vytvorí uzol v strome. Algoritmus hľadá dva uzly s najmenšou váhou, ktoré nahradí interným uzlom, ktorý bude mať svoju váhu nastavenú na súčet váh nájdených uzlov a za deti (synov) interného uzla budú nastavené tieto dva uzly. Problém nastáva v momente, keď je v strome len jeden uzol a algoritmus nemôže začať svoju činnosť. Tento problém je v implementácii vyriešený detekciu tohto prípadu a následným pridaním extra uzlu do stromu. Toto riešenie zaručí správny beh algoritmu a nič nezmení na dĺžke kódového slova pôvodného uzla. Po skončení behu algoritmu sa v metóde `save_codes` ukladajú kódové slová pre znaky v strome. Následne prebieha prevod kódových slov na kanonický tvar (kanonocký Huffmanov kód) v metóde `canonicalize`. Metóda `encode` vytvára hlavičku a komprimuje bajty vstupného súboru. Prvý bajt hlavičky reprezentuje počet rôznych dĺžok kódových slov. Druhý bajt zachytáva dve informácie. Horné tri bity informujú, koľko bitov je výplň v poslednom bajte súboru (dáta sa vždy ukladajú v bajtoch, no po skončení komprimácie nemusí byť počet bitov násobok ôsmich). Spodné tri bity slúžia na potenciálne zníženie veľkosti hlavičky, ktorá je možná v niektorých prípadoch. Informujú, koľko dĺžok kódových slov od začiatku postupnosti je nulových (tj. žiadny znak nie je reprezentovaný kódovým slovom s danou dĺžkou), inak povedané, je to posun v postupnosti dĺžok na prvú dĺžku, na ktorú sa kóduje nejaký znak. Nasleduje sekvencia bajtov, kde sa ukladá informácie, koľko znakov je kódovaných na určitú dĺžku kódového slova. Prípad, keď je v súbore rovnomerne rozložených všetkých 256 znakov ($l_8 = 256$), je riešený na strane dekódera. Pomocou hodnoty vyššie spomenutého offsetu vie dekóder odhaliť a opraviť na ôsmich bitoch pretečenú pôvodnú hodnotu 256, tj. po pretečení 0, späť na hodnotu 256. Poslednou časťou hlavičky sú znaky vstupnej abecedy. Po vytvorení hlavičky dochádza k kódovaniu bajtov vstupného súboru a následne zápis kódových slov pre tieto bajty do súboru. V prípade potreby výplne v poslednom bajte sa doplňajú nulové bity. Pri dekódovaní sa najskôr prečíta hlavička zo vstupného súboru. Následne sa dekóduje súbor použitím algoritmov/metód FCFS a FastCHD. Implementácie týchto algoritmov sa nachádzajú v metóde `decode`.

5 Adaptívne Huffmanove kódovanie

Adaptívne Huffmanove kódovanie je v implementácii reprezentované triedou `adaptive_huffman`, ktorá je v súbore `adaptive_huffman.cpp`. Trieda implementuje algoritmus FGK. V konštruktore triedy sa vytvára strom s jediným uzlom, tzv. NYT uzol. V metóde `encode` sa každy bajt vstupného súboru kóduje pomocou binárneho stromu a následne prebieha aktualizácia stromu. Metóda `decode` je zrkadlová, vstupné bity sa dekódujú na znaky pomocou binárneho stromu. Po každom dekódovanom znaku prebieha aktualizácia stromu. Problém výplne posledného bajtu nastáva aj u adaptívneho Huffmanovho kódovania. Implementované riešenie používa posledný bajt na uloženie informácie o počte bitov slúžiacich ako výplň predchádzajúceho bajtu. V prípade, že počet bitov výplne je 0 a hodnota posledného bajtu je viac ako 7, je možné ušetriť bajt pre informáciu o výplni. Dekóder vie, že počet bitov skutočnej výplne môže byť maximálne 7 bitov. Ak zistí, že hodnota posledného bajtu je viac ako 7, vie, že posledný bajt nedržal informáciu o výplni, tj. počet bitov výplne je v skutočnosti 0. Aktualizácia stromu je implementovaná v metóde `update_tree` a odpovedá logike aktualizácie stromu v algoritme FGK. Profilovaním bolo odhalené, že najviac času pri kódovaní/dekódovaní tvoria práve volania tejto funkcie. Pre zrýchlenie činnosti kódera/dekódera je nevyhnutné optimalizovať práve túto metódu. Po prvotnej implementácii bola metóda prepísaná do značne efektívnejšie podoby, čo potvrdili moji priebežné merania.

6 Model - diferencia pixelov

Model je implementovaný v súbore `pix_diff_model.cpp`. Obsahuje dve funkcie pre účely samotnej transformácie a reverznej transformácie. Transformácia prechádza každý bajt vstupných dát, a novú hodnotu bajtu počíta ako rozdiel hodnoty aktuálneho bajtu a hodnoty predchádzajúceho bajtu. Reverzná transformácia funguje opačne, tj. správnu hodnotu získava pričítaním hodnoty predchádzajúceho bajtu k hodnote aktuálneho bajtu.

7 Vyhodnotenie

Testovanie implementácie prebiehalo priebežne, a to na systéme Ubuntu LTS 18.04 a na školskom serveri *merlin*. Merania, experimenty prebiehali s obrazovými dátami priloženými k zadaniu projektu na školskom serveri *merlin*. Program bol preložený prekladačom GCC 8.3. Všetky vstupné `raw` súbory mali rovnakú veľkosť, a to 262144 bajtov. Pre každú dvojicu - (súbor, metóda) - bolo vykonaných 10 meraní a výsledná hodnota sa získala spriemerovaním týchto hodnôt. Testované metódy boli nasledovné:

- Statické Huffmanovo kódovanie (kanonický Huffman)
- Statické Huffmanovo kódovanie (kanonický Huffman) s modelom (diferencia pixelov)
- Adaptívne Huffmanovo kódovanie (FGK)
- Adaptívne Huffmanovo kódovanie (FGK) s modelom (diferencia pixelov)

	Stat. Huffman	Stat. Huffman + model	Adapt. Huffman	Adapt. Huffman + model
hd01.raw	127140	111607	127271	111694
hd02.raw	121401	109196	121547	109304
hd07.raw	183920	126375	184048	126449
hd08.raw	138826	115549	138963	115620
hd09.raw	218228	153538	218430	153637
hd12.raw	203170	143900	203357	143988
nk01.raw	213272	198879	213409	198969

Tabuľka 1: Veľkosť zakomprimovaného súboru (bajty) pre každý raw súbor

	Stat. Huffman	Stat. Huffman + model	Adapt. Huffman	Adapt. Huffman + model
hd01.raw	0,019	0,019	0,020	0,018
hd02.raw	0,015	0,016	0,020	0,019
hd07.raw	0,020	0,022	0,026	0,020
hd08.raw	0,018	0,019	0,021	0,019
hd09.raw	0,020	0,019	0,031	0,024
hd12.raw	0,017	0,015	0,028	0,022
nk01.raw	0,019	0,018	0,029	0,028

Tabuľka 2: Čas komprimácie (sekundy) pre každý raw súbor

	Stat. Huffman	Stat. Huffman + model	Adapt. Huffman	Adapt. Huffman + model
hd01.raw	3,88	3,41	3,88	3,41
hd02.raw	3,70	3,33	3,71	3,36
hd07.raw	5,61	3,86	5,61	3,86
hd08.raw	4,23	3,52	4,24	3,52
hd09.raw	6,65	4,68	6,66	4,69
hd12.raw	6,20	4,39	6,20	4,39
nk01.raw	6,50	6,07	6,51	6,07

Tabuľka 3: Priemerný počet bitov potrebných na zakódovanie bajtu obrazu pre každý raw súbor

	Stat. Huffman	Stat. Huffman + model	Adapt. Huffman	Adapt. Huffman + model
bitov na bajt	5,25	4,18	5,25	4,19

Tabuľka 4: Priemerný počet bitov potrebných na zakódovanie bajtu obrazu pre každú metódu

	Stat. Huffman	Stat. Huffman + model	Adapt. Huffman	Adapt. Huffman + model
čas (sekundy)	0,018	0,018	0,025	0,021

Tabuľka 5: Priemerný čas potrebný na komprimáciu súboru pre každú metódu

8 Návod k použitiu

Pomocou príkazu `make` v priečinku so zdrojovými súbormi sa vykoná zostavenie programu. Predpokladá sa, že cieľový systém obsahuje prekladač GCC s podporou C++11, tj. GCC 4.9 a novšie. Preklad prebieha s optimalizáciami za účelom za účelom rýchleho kódovania/dekódovania súborov programom. Zostavený program má názov `huff_codec`. Príkazom `make debug` sa vytvára program vhodný na ladenie, tj. neoptimalizovaný, obsahujúci čo najviac informácií pre ladenie. Odstránenie vytvoreného binárneho súboru a objektových súborov je možné pomocou príkazu `make clean`. Samotný program má nasledovné možnosti:

- `-c` - komprimovať súbor
- `-d` - dekomprimovať súbor
- `-i` - vstupný súbor
- `-o` - výstupný súbor
- `-h static` - použiť statické Huffmanove kódovanie
- `-h dynamic` - použiť adaptívne Huffmanove kódovanie
- `-m` - použiť model diferencie pixelov na spracovanie (*preprocessing, postprocessing*) dát
- `-h` - zobraziť pomocník k programu

Príklady spustenia:

- `./huff_codec -h static -c -i obrazok.raw -o komprimovany_obrazok.raw -m`
  Program predspracuje dáta zo súboru *obrazok.raw* pomocou modelu a následne takto predspracované dáta zakomprimuje pomocou statického Huffmanovho kódovania. Finálne dáta sa uložia do súboru *komprimovany_ obrazok.raw*.
- `./huff_codec -h adaptive -d -i komprimovany_obrazok.raw -o obrazok.raw -m`
  Program dekomprimuje dáta zo súboru *komprimovany_ obrazok.raw* pomocou adaptívneho Huffmanovho kódovania. Dekomprimované dáta sú následne spracované reverznou podobou modelu. Finálne dáta sa uložia do súboru *obrazok.raw*.

9 Informačné zdroje

- *Úvod do problematiky kódování a komprese dat*
  `https://wis.fit.vutbr.cz/FIT/st/cfs.php?file=%2Fcourse%2FKKO-IT%2Flectures%2FKKO-01.pdf&cid=12830`
- *Statistické metody komprese dat (Tunstallovo, Huffmanovo a Aritmetické kódování)*
  `https://wis.fit.vutbr.cz/FIT/st/cfs.php?file=%2Fcourse%2FKKO-IT%2Flectures%2FKKO-04.pdf&cid=12830`
- *Adaptive Huffman coding - FGK*
  `http://www.stringology.org/DataCompression/fgk/index_en.html`
- *Faller, Gallager, Knuth = Algorithm FGK*
  `https://sites.google.com/site/compgt/fgk`