FAKULTA
APLIKOVANÝCH VĚD
ZÁPADOČESKÉ
UNIVERZITY
V PLZNI

Huffmanovo kódování

Semestrální práce z KIV/TI

Patrik Harag

Jakub Vašta

2016-12-07

Obsah

Zadání

Vytvořte jednoduchý kompresní a dekompresní program. Jeho vstupem bude obecný (binární) soubor, výstupem soubor komprimovaný Huffmanovým kódováním. Výstupní soubor v sobě musí obsahovat všechny potřebné informace pro dekompresi, tj. zejména četnosti vstupních znaků. Pro jednoduchost předpokládejte, že vstupní soubor bude dlouhý maximálně 2 GiB.

Program se bude volat z příkazové řádky, např. pro kompresi

```
huffman -c input.txt compressed.huf
```

nebo pro dekompresi

```
huffman -d compressed.huf decompressed.huf
```

Po kompresi nechť program vypíše statistiku: jaká byla velikost vstupního souboru, jaká je velikost výstupního kódu (bez četností a dalších informací nutných k dekompresi), jaká je velikost pomocných informací, jaký je dosažený kompresní poměr.

Kvůli sobě (i kvůli mně) do programů zabudujte přepínač "-v". Pak bude program čitelným způsobem hlásit, co dělá či udělal. Například po spuštění

```
huffman -v -c input.txt compressed.huf
```

by program mohl hexadecimálně vypsat vstup (předpokládá se, že krátký), vypsat tabulku četností, kódovací tabulku (vstupní symbol ⇒ binární kód) a nějakým srozumitelným způsobem generovaný výstup. Obdobně by informace užitečné pro kontrolu činnosti program vypisoval při dekompresi.

Zájemci, kteří si chtějí se zadáním pohrát, mají nejméně dvě možnosti. První je vymyslet efektivní uložení dodatečných informací nutných k dekompresi, neboť ty budou hrát u krátkých souborů významnou roli. Druhá je vyzkoušet, jaký vliv má volba vstupní abecedy na kompresní poměr; první volba asi padne na osmice bitů (tj. vstupní abeceda má 256 symbolů). Jak se chování kompresoru změní, bude-li například vstupním symbolem čtveřice bitů (půlbyte, tj. vstupní abeceda má 16 symbolů) nebo šestnáctice bitů (dva byte, tj. vstupní abeceda má 65536 symbolů)? Alternativně je možné zpracovat zadání "Adaptivní Huffmanovo kódování".

Analýza

Problém Huffmanova kódování je možné rozdělit do těchto čtyř částí:

1. Načítání ze vstupního souboru
2. Vytvoření tabulky četností a kódovacího stromu
3. Zakódování vstupního souboru a jeho zapsání do výstupního souboru
4. Dekódování zakódovaného souboru

Načítání ze vstupního souboru

V této fázi je potřeba se rozhodnout po jak velkých částech ze souboru načítat, první možnost, kterou jsme využili, je načítání po bytech.

Vytvoření tabulky četností a kódovacího stromu

Soubor ke kódování budeme muset procházet dvakrát, poprvé abychom zjistili četnost znaků vstupní abecedy a podruhé abychom dané znaky zakódovali. Z tabulky četností vytvoříme kódovací strom, který má ve svých listech uloženy kódovací značky.

Zakódování vstupního souboru a zapsání do výstupního souboru

Jak jsme již napsali, při druhém průchodu načítané byty kódujeme a zapisujeme do výstupního (zakódovaného) souboru opět po bytech. Vyvstává zde problém zarovnání na byte. Tento problém řešíme doplněním nevýznamových 0. Do výstupního souboru také musíme uložit informace k dekódování.

Dekódování zakódovaného souboru

Z informací k dekódování opět sestrojíme Huffmanův strom pro naše kódování. Načítáme znaky ze souboru a hledáme ve stromě jaký znak ze vstupní abecedy náleží této kódovací značce. Na konci souboru si musíme dát pozor na nevýznamové 0, které jsme přidali pouze kvůli zarovnání na byte.

Implementace

Kódování

Tvorba Huffmanova stromu

Algoritmus funguje tak, že pro všechny kódové značky o frekvenci větší než nula vytvoří uzel, do uzlu se uloží i jeho frekvence. Všechny tyto uzly uloží do seřazeného seznamu nebo haldy (prioritní fronty), klíčem řazení je frekvence.

Následně ze seznamu vybereme (a odebereme) dva uzly s nejmenší frekvencí a přidáváme nový uzel, který obsahuje reference na vybrané uzly a má frekvenci rovnou součtu frekvencí vybraných uzlů. Opakujeme tak dlouho, dokud nezbude jediný uzel. Poslední uzel je kořenem Huffmanova stromu.

```
public static Node build(int[] frequencies) {
    Comparator<Node> comparator = Comparator.comparingInt(Node::getFrequency);
    PriorityQueue<Node> queue = new PriorityQueue<>(comparator);

    for (int i = 0; i < frequencies.length; i++) {
        if (frequencies[i] > 0) {
            queue.offer(new LeafNode(frequencies[i], i));
        }
    }

    while (queue.size() != 1) {
        Node a = queue.poll();
        Node b = queue.poll();

        queue.offer(new TreeNode(a, b));
    }

    return queue.poll();
}
```

Zakódování vstupního souboru

Pro kódování jsme si Huffmanův strom převedli do pole, indexy jsou od 0..255. Kódovou značku pro daný byte získáme na indexu, který je roven hodnotě daného bytu v desítkové soustavě.

Na začátek komprimovaného souboru ukládáme tabulku četností pro pozdější dekódování. Následně procházíme vstupní soubor po bajtech, jednotlivé bajty zakódujeme a uložíme do bufferu, pokud je v bufferu alespoň bajt, zapíšeme ho do výstupního souboru.

Struktura binárního formátu komprimovaného souboru je následující:

- 4 bajty pro velikost tabulky četností
- *velikost abecedy* × 4 bajty pro tabulku četností – ukládá se pouze četnost
- komprimovaná data + zarovnání na bajt

Dekódování

Ze zkomprimovaného souboru získáme tabulku četností a stejným způsobem, jako v případě kódování sestrojíme Huffmanův strom.

Z tabulky četností je zjištěn celkový počet bajtů, které mají vzniknout. Pomocí této hodnoty je následně kontrolováno, aby nedošlo i k dekódování nevýznamových nulových bitů na konci souboru, které slouží k zarovnání souboru na celý bajt.

Komprimovaný soubor dále načítáme opět po bytech. Bajty rozdělujeme na bity a na základě hodnot se pohybujeme v Huffmanově stromě. Pokud narazíme na list, zjistíme jeho hodnotu a tu uložíme do výstupního souboru.

Závěr

Výkon

Výkon programu na procesoru Intel Core i5-4570

Vstupní soubor	**Velikost souboru**		**Kompresní poměr**	**Výsledný čas**	
	originální	**komprimovaná**		**komprese**	**dekomprese**
Textový soubor	36 kB	25 kB	0,31	29 ms	22 ms
Textový soubor	6,61 MB	3,63 MB	0,45	570 ms	280 ms
RAW fotografie	257 kB	238 kB	0.07	90 ms	45 ms
PDF dokument*	4,44 MB	4,37 MB	0,02	10,6 s	316 ms
Video (mp4)*	53,7 MB	53,7 MB	0	4,6 s	3,0 s

* tyto formáty jsou samy o sobě komprimované

Zajímavé je, že pro různé druhy dat s podobnou velikostí vstupu je komprese různě rychlá. Dekomprese ale bude vždy rychlejší než komprese, protože jsme ji realizovali, na rozdíl od komprese, pomocí bitových operací.

U malých souborů zabírají údaje pro dekódování více místa, než samotná komprimovaná data. Může se tak stát, že výsledný komprimovaný soubor bude mnohem větší, než původní soubor. Program je tedy vhodný pouze pro větší soubory (10 kB a více).

Možná vylepšení

1. Obdobně jako dekódování by se dalo optimalizovat kódování - místo používání řetězců používat bit-sety a bitové operace.
2. Zkusit načítat ze vstupního souboru po jiných částech než po bytu.

3. Omezit průchody vstupním souborem na jeden a tím přejít na Adaptivní Huffmanovo kódování.

Obsah odevzdávaného archivu

V odevzdávaném archivu se nachází celý projekt z vývojového prostředí NetBeans. Kořenový adresář obsahuje připravené ukázkové soubory a dávky pro jejich kompresi a dekompresi. Spustitelný jar se nachází ve složce *dist*.