Алгоритъм На Хъфман за компресия на данни. честотни таблици

Проект по системи за паралелна обработка

Изготвил: ПЕТЪР КАМБУРОВ

ф.н: 80840

Компютърни науки, 3 курс, 7 група

Ръководител:

АС. ХРИСТО ХРИСТОВ

Дата: 27.06.2015 Проверил:.................................

(ас. Христо Христов)

# Въведение

Алгоритъмът на Хъфман, разгледан тук е сравнително прост универсален алгоритъм за

компресия без загуба на данни. При него се предполага, че е даден краен поток от числа в някакъв предварително фиксиран интервал. Ще считаме, че става дума за символи, кодирани със ASCII код, т.е. ще разглеждаме информацията като поредица от байтове (числа в интервала 0..255). Алгоритъмът се базира на простата идея, че най-често срещаните символи в поредицата трябва да се записват с наймалък брой битове. Така той построява нова азбука, която следва тази идея и след това превежда информацията в новата азбука. Кодирането е обратимо, тоест по кодираната последователност може да се декомпресира да се намери първоначалната поредица.

# Условие на проекта

Разглеждаме алгоритъмът на Хъфман в частта му свързана с построяване на честотна таблица на входния поток от информация. Задачата е да се напише програма, която строи честотна таблица на даден двоичен или текстов (достатъчно голям) файл. Програмата да разпределя по подходящ начин работата за построяване на честотната таблица между две или повече нишки (задачи);

# Алгоритъм на Хъфман

## Компресиране

Нека трябва да компресираме даден низ от символи. Искаме да построим двоично дърво, от което ще определим азбука за компресиране. Алгоритъмът за построяване на дърво се състои от следните стъпки:

1. Създава се честотна таблица на низа за всеки символ се записва броят на срещанията му.
2. Нека различните символи в низа са n на брой. Създаваме n дървета от по един елемент, където всяко дърво съдържа символ и броя на срещанията му.
3. Намираме двете дървета, които в корените имат най-малко число. Обединяваме дърветата в ново дърво, като в корена записваме сумата от стойностите в двете намерени дървета.
4. Повтаряме стъпка 3, докато не получим само едно дърво дървото на Хъфман за дадения низ.

## Декомпресиране

Декомпресирането на данните става лесно при условие, че имаме дървото на Хъфман. Вървим едновременно по двоичния низ и по дървото, като всеки път като срещнем 0 завиваме наляво, а при 1 надясно. Когато стигнем до листо, записваме съответния символ и рестартираме от корена. Така стъпка по стъпка получаваме първоначалния низ.

# Стартиране на проекта

* Чете името на входния файл от подходящо избран команден параметър – например “f file.dat”;
* Втори команден параметър задава максималния брой нишки (задачи) на които разделяме работата по построяването на честотната таблица – например “t 1” или “tasks 3”;
* Стартиране на графичния потребителски интерфейс става като се зададе само един параметър – “-g”.

# Описание на реализираните алгоритми

## Честотна таблица

След известно проучване и тестове установих, че няколко неща:

* Операционната система прави много добри оптимизации и поради тази причина последователното прочитане на даден файл е най-бързия начин за четене. Можем да реализираме алгоритъм, който да разпределя нишките да четат различни части от файла, но това няма да ни повиши производителността.(дори ще забави нещата)
* „Bottle Neck“-а е в броенето кой символ колко пъти се среща.

Именно поради тези причини реализирах следния **Producer/Consumer** алгоритъм:

* **Една нишка отговаря за четенето на файла**. Тя прочита определен масиви байтове(по подразбиране с размер 2048) и ги записва в блокираща опашка(**BlockingQueue**) с максимален размер 200 масива.(т.е. имаме нужда от 2048\*200 байта памет, разбира се тези числа за изкарани като константи и могат да бъдат променени по всяко време). Тези числа са избрани случайно – с други е възможно да имаме по-добра производителност.
* **Всички останали нишки имат за цел да обработват тези масиви от байтове в BlockingQueue.** Всяка нишка „си взима“ един масив, и се грижи за попълването на хеш таблица(<символ, брой срещания >). При целия този процес е направен така, че да бъде ***thread-safe***.

**Важна забележка:**

Реализирал съм отделна функция, която се грижи за случая когато имаме **една нишка**. Можех да направя, така че да работи с описания алгоритъм, но не алгоритъма става безсмислен – ще чета файла с една нишка, ще пиша в опашка и после ще обработвам записаното и този процес ще се повтаря многократно. Просто целия смисъл от писането в опашка се губи, така че за класически случай(този с една нишка) приемам случая да мина през файла и да записвам в хеш таблица всеки символ и броя на срещанията му. В крайна сметка ще получа „по-слаби“ статистики за ускорение и ефективност, но не намирам смисъл умишлено да забавям работата с една нишка, за да се повишат тези статистики.

## Създаване на дървото на Хъфман

Чрез честотната таблица построяваме дървото на хъфман по описания в раздел 3 алгоритъм.

## Построяване на таблица на кодирането

Чрез обхождане на дървото на Хъфман получаваме пътя от корена до листото(а листата са символите във файла). Този път се пази в структурата ***CodeSymbol***, тя се състои от 2 полета ***long Code*** и ***int length***.

Например ако символът „А“ се кодира с 01, то CodeSymbol ще има вида

code: 00000000 00000000 00000000 00000001, length: 2

Има възможност ***length*** да стане по-голямо от 64 и тогава ще имаме проблем(но за достатъчно големи файлове не мисля, че ще имаме проблем – при 100МБ файл length нормално е около 10-15, разбира се това зависи от самите символи във файла).

## Компресиране и декомпресиране

След като сме създали таблицата на кодирането, четем оригиналния файл и кодираме всеки символ с съответните битове. За целта ще „запълваме“ 64-те бита в типа long и ще го записваме в компресирания файл.

Например ако А се кодира с 1111 1111, а B се кодира с 0000 0000 и трябва да компресираме файл със съдържание **ABAB**, то компресирания файл ще е:

1111 1111 0000 0000 1111 1111 0000 0000, което е точно един long. Но ако при същите кодирания трябва да кодираме **ABABA**, то ще трябва да запишем 2 long-а

1-ви long: 1111 1111 0000 0000 1111 1111 0000 0000

2-ри long: 1111 1111 **0000 0000 0000 0000 0000 0000**

Но възниква един проблем при декодиране - подчертаните нули всъщност декодират AAA, т.е. ще получим ABABA**AAA, а не ABABA.** За да избегнем такива грешки, ще записваме броят на символите в началото на кодирания файл.Т.е.

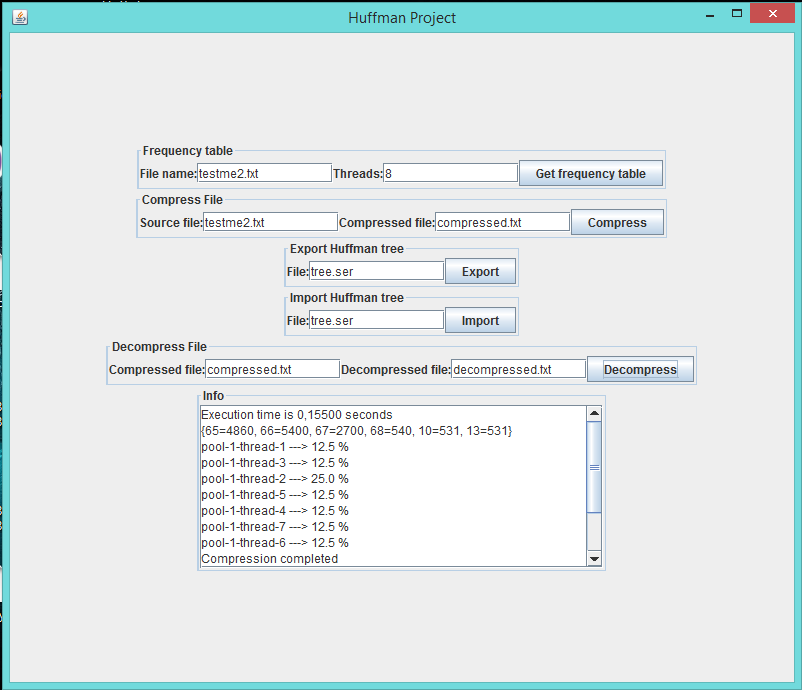
1-ви long: 0000 0000 0000 0000 0000 0000 0000 0101 = **5**

2-ри long: 1111 1111 0000 0000 1111 1111 0000 0000

3-ти long: 1111 1111 **0000 0000 0000 0000 0000 0000**

Така ще знаем, че този файл съдържа 5 символа и след като сме ги прочели значи целият файл е декомпресиран(нулите в края ще останат непрочетени). Самото декомпресиране от битове в символ се извършва като обхождаме дървото на Хъфман по начина описан в раздел 3.

# Функционалности. Графичен интерфейс.



* ***Get frequency table*** – След задаване на файл и брой нишки можем да намерим честотната таблица. Извежда се времето, необходимо за изчислението на таблицата, самата таблица и статистика за нишките – коя нишка колко % от работата е свършила.
* ***Compress*** – Позволява ни да компресираме даден файл след като вече сме му намерили честотната таблица.
* ***Export Huffman tree*** – Може да запазим дървото на Хъфман (използва сериализация).
* ***Import Huffman tree*** – Може да заредим дървото на Хъфман от даден файл, и след това веднага да десериализираме с негова помощ.
* ***Decompress*** – Декомпресиране на вече компресиран файл.
* ***Info*** – Панел за информация.

# Тестови показатели. Диаграми на ускорение и ефективност.

Разработено е приложение на JAVA и е тествано на 4 ядрен мултипроцесор с цел да се оценят ускорението ***S*** и ефективността ***Е*** на описания алгоритъм, където ако ***T(p)*** е времето необходимо за завършване на работата на алгоритъм с ***p*** на брой нишки то:

***S(p) = T(1)/T(p)***

***E(p) = S(p)/p***

Тестовете са направени върху файл с големина: **205MB**.

Процесор: ***I7-3610QM, 4 ядрен***

Рам: ***4 GB***

Тестови показатели(в секунди).

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *Number of threads* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* |
| execution 1 | 11,569 | 9,593 | 8,155 | 6,977 | 6,563 | 6,827 | 6,427 | 6,521 |
| execution 2 | 9,484 | 9,767 | 7,938 | 7,127 | 6,521 | 6,709 | 6,712 | 6,84 |
| execution 3 | 8,532 | 9,604 | 7,898 | 7,588 | 6,707 | 6,386 | 6,434 | 6,333 |
| execution 4 | 8,911 | 9,672 | 8,044 | 7,381 | 7,015 | 6,733 | 6,718 | 6,721 |
| execution 5 | 8,958 | 9,664 | 8,061 | 7,037 | 6,929 | 6,729 | 6,801 | 6,618 |
| ***Average*** | **9,4908** | **9,66** | **8,0192** | **7,222** | **6,747** | **6,6768** | **6,6184** | **6,6066** |
| ***Ускорение*** | **1** | **0,9825** | **1,18351** | **1,3142** | **1,40667** | **1,4215** | **1,434** | **1,436563** |
| ***Ефективност*** | 1 | 0,4912 | 0,3945 | 0,3285 | 0,28133 | 0,2369 | 0,20486 | 0,17957 |