Документация по проект

По СДП

Тема 2: Хъфман-компресиране.

Изготвил:

Виктор Василев ф.н 45676

Информатика, 2-ри курс

Група 3

12.08.2021г.

Съдържание:

1. [Описание на заданието. Какво е Хъфман компресиране. …….](#book1)
2. [Реализация на Функционалностите. Инструкции за ползване…………………………………………………………………………………..](#book2)

* [Интерфейс от командния ред……………………………………………..](#book21)
* [Създаване на архив по даден списък от файлове и директории…………………………………………………………………………..](#book22)
* [Разархивиране на един или всички файлове от архива……](#book23)
* [Извеждане на списък на компресираните файлове с информация за нивото на компресия…………………………………](#book24)
* [Проверка за повреден архив………………………………………………](#book25)
* [Обновяване на вече архивиран файл с нова версия на оригиналния…………………………………………………………………………](#book26)

1. [Използвани ресурси, материали и технологии………………………](#book3)
2. Описание на заданието. Какво е Хъфман компресиране.

Компресирането на данни е важен дял от информатиката, който дава възможност да съхраняваме данни използвайки по-малко памет отколкото е предназначена за самите данни. Алгоритмите за компресиране помагат за по-бързия трансфер на данни и редуциране на мястото им за съхранение. Компресирането се дели на два вида – компресиране със загуба (lossy compression) и компресиране без загуба (lossless compression). Алгоритъмът за компресиране на Хъфман компресира данните без загуба, което го прави ефективен при компресиране на данни от типа на изпълними файлове, текстови файлове, таблици и др.

Алгоритъмът на Хъфман е от графата алчни алгоритми. Целта на алгоритъма е да съпостави на всеки символ в съобщението по-кратък двоичен код. Как го постига? Алгоритъма строи двоично дърво, чиито листа представляват символите в съобщението, което искаме да компресираме. Колкото по-често-срещан е даден символ в съобщението, толкова по-близко е той до корена на дървото, което гарантира по-бърз достъп и съответно по-малко памет за кодирането му.

1. Реализация на Функционалностите. Инструкции за ползване.

2.1 Интерфейс от командния ред.

Интерфейсът от командния ред действа като входно-изходно устройство. Той обработва командите от потребителя и извежда съобщение за изпълнението на програмата. Описан е във файла interface.cpp. Тук също се обработват и изключенията, хвърлени от класовете и се връщат съобщения за грешка. Няма особености по реализацията.

Тук е момента да спомена че реализираната програма не е предназначена за архивиране и разархивиране на файлове по-големи от 4GB, защото използвам 32-битови типове за съхранението на размери на файлове.

Команди: archive, extract one/all, info, check, update.

* 1. Създаване на архив по даден списък от файлове и директории.

Извършва се чрез командата archive, след което се подава списък от файлове и директории, като се изписва пълния път на файловете в системата. **Входният формат е както следва:**

[път до папка][файл]\*[файл]\*[файл] … \*[файл]?[път до папка][файл]\*[файл]…

Файловете се разделят със „\*“ , а различните пътища с „?“.

Пример: C:\Users\UserName\Desktop\Project\Files\src\program.cpp\*program2.java?C:\Users\UserName\Downloads\book.txt\*document.doc

Архивът ще съдържа четири файла.

След това се подава път и име на архивния файл, заедно с разширението му, като то няма значение. При подаване на вече съществуващ файл той може да бъде презаписан с новото съдържание на архивиран от програмата файл. Целия процес на архивиране се управлява от класа Encoder.

**Един файл се архивира по следния начин:**

В масив freq се записва колко срещания има всеки байт (unsigned char) във файла. Той е с размер броят на всички възможни байтове, а именно 2^8=256 общо символа.

След това построяваме хъфмановото дърво:

Представяме всеки символ като възел, съдържащ колко пъти се среща. Съхраняваме възлите във min heap (приоритетна опашка).

Изваждаме двата най-малки възела от опашката, създаваме нов възел с тях като ляво и дясно дете и стойност сбора от техните срещания. Вкарваме новия възел обратно в опашката. Повтаряме операцията докато не остане един единствен възел и той ще бъде корен на хъфмановото дърво. Само листата имат съпоставен символ.

Следва на всеки символ да съпостaвим съответстващият му хъфманов код:

Започваме обхождане (DFS) от корена на дървото. Ако изберем да продължим към лявато дете, записваме 0, дясното – 1. Когато стигнем до листо на дървото, прочитаме символа и му съпоставяме получения двоичен код. По-често срещаните символи ще имат по-кратък двоичен код, например от 3, 4 или 5 бита (вместо 8). Някои символи може да се получат с по-голям хъфманов код, дори от порядъка на 13 – 14 бита, но това е ОК защото те се срещат сравнително рядко.

Класа bitVector реализира структура наподобяваща std::vector<bool>, която обаче позволява записването и във файл без да губим от битовете. Състои се от std::bitset порции по 32 бита и има първоначален капацитет общо 4KB. Идеята е когато размера достигне разликата на капацитета и максималната дължина на кодиран символ (дълбочината на дървото), векторът да се изпразни като запише всичко в архивния файл и да почне да се попълва наново.

Тъй като операцията push\_back на bitVector добавя само по един бит, а максималната единица, която можем да запишем във файл е байт, има остатъчни битове във вектора, които допълваме до байт и след това записваме остатъка във файла.

Дървото също записваме във bitVector преди да го запишем в архивния файл. Записваме го по следния начин – обхождаме рекурсивно дървото, ако се натъкнем на възел който не е листо записваме бит 1 и обхождаме лявото и дясното поддърво. Ако се натъкнем на листо, записваме бит 0 и записваме 8-те бита на символа който представлява. От този запис следва горната граница за размер на дървото в паметта. Всяко листо ще заема 8+1 = 9 бита, а всяко нелисто – 1 бит. Тъй като хъфмановото дърво е пълно двоично дърво, при n на брой символа във файла ще имаме n листа и n-1 нелиста, т.е

9\*n + n – 1 = 10n – 1 бита. При n = 256 това са 2559 бита или когато допълним до точен байт – 320 байта.

Преди да запишем дървото във файла записваме неговия размер в битове като 4 байтово число, след него записваме флаг ЕОТ (end of tree) чието прочитане при резархивирането ни гарантира че сме прочели цялото дърво и само него успешно и можем да продължим нататък.

Така горната граница за запис на дърво се получава 320 + 4 + 1 = 325 байта.

Горната граница на размера на кодирания файл от друга страна е близка до размера на оригиналния файл преди компресията. Това се получава защото съществуват файлове които вече са минали през достатъчно ефективна компресия за да не могат да намалят размера си чрез хъфман кодиране. Така ако се опитаме да компресираме такъв файл, размера на архивния файл може да бъде дори по-голям от този на оригиналния.

Така файлът се компресира до дърво (макс = 325 байта) + кодиран файл(макс ~ първоначален размер).

**Как се компресират множество файлове:**

При компресирането на папки използвам библиотекката std::filesystem. Тя позволява да обходим структурата на папка рекурсивно в дълбочина чрез recursive\_directory\_iterator и да извличаме информация за елементите. Пътищата за всеки файл от началната директория до самия него запазваме в отделен низ string, след което сортираме тези низове по имената на файловете. Това ни позволява по-лесно търсене на един файл при резархивиране по-късно. След като са сортирани ги обединяваме в един общ низ с разделител м/у тях и кодираме този низ с алгоритъма на Хъфман. По този начин запазваме структурата на директориите и имената на файловете в компактен вариант. Записваме дървото и кодираните пътища във файла. Записваме крайната позиция на пътищата в архивния файл и размера на целия низ в самото начало на архивния файл. След това кодираме един по един всеки файл по посочения по-горе начин. Следва да запишем метаданни за файловете непосредствено след информацията за пътищата с имената, като предварително сме запазили място памет за това точно преди самите кодирани файлове. Метаданните съдържат: брой файлове, за всеки файл – размер на файла, начална позиция в архивния файл, чексума, крайна позиция в архивния файл. (Описание за самата чексума има по-нататък). Като метаданните за всеки файл са също подредени в сортиран ред както пътищата им. След като приключим със всички файлове, изчисляваме чексумата на целия архивиран файл и я записваме в края (4 байта).

Финалната структура на архивния файл е:

[pathsEndPos].[sizeOfpaths].[treeSize].[tree].[EOT].[pathsMeta].[filesCount].[filesMetaData].[compressedFiles].[checksum]

filesMetadata => [fileSize][startPos][checksum][endPos] за всеки

compressedFiles => [treeSize][tree][eot][file] за всеки.

* 1. Разархивиране на един или всички файлове от архива

Извършва се чрез командата extract, след което се указва един или всички фалове чрез one/all, подава се път към архивен файл, след което път към дестинация за разархивиране. Ако добавите последна папка която не съществува, папка с такова име ще бъде създадена.

Разархивирането е имплементирано по следният начин – прочитаме крайната позиция на пътищата в архива, прочитаме техния размер, прочитаме дървото, декодираме пътищата като един символен низ с рамера, който сме прочели, разделяме низа на отделни низове за всеки файл, прочитаме и съхраняваме метаданните, за всеки архивиран файл – прочитаме дървото му и спрямо него и пътя на файла, го разархивираме в посочената дестинация, запазвайки структурата на папките където се намира.

Данните архивния файл отново се съхраняват във bitVector с капацитет 4KB след което четем битовете от вектора. Щом стигнем до позиция във вектора по-голяма от разликата на капацитета и дълбочината на дървото отново следва да освободим паметта и да прочетем следващите 4KB.

**Как се прочита дърво от файл:**

Прочита се големината на дървото в битове. Записват се битовете в bitVector с капацитет големината на дървото в байтове. След това използваме рекурсивен алгоритъм за прочитането на дървото от паметта. Прочитаме следващия символ. Ако е флагът EOT означава че сме прочели правилно цялото дърво и само него. След това дървото се записва в архивния файл.

**Как се съхраняват метаданните:**

Чрез структората fileInfo, която съдържа следните член данни: път – съхранява структурата на директории до файла, име – играе ролята на ключ за търсене, размер на оригиналния файл – указва колко символа трябва да бъдат декодирани, чексума – предоставя информация за непокътността и идентичността на файла, начална позиция – от къде започва кода на файла в архивния файл, крайна позиция – къде свършва.

fileInfo обектите се записват във вектор чиито капацитет се задава като брой файлове. Знаем че данните са сортирани, тъй като сме ги записали сортирани в архивния файл.

**Разархивиране на всички файлове:**

След прочитането и съхранението на метаданните, за всеки файл, започваме от началната позиция и декодираме толкова символи, колкото е размера на файла. Папките описани в пътя се създават в процеса (ако вече не съществуват) като започваме от папката посочена за дестинация.

**Разархивиране на един файл:**

Аналогично, но първо намираме обекта с информацията му с двоично търсене. Отиваме на началната позиция на файла и разархивираме само него в посочения път.

* 1. Извеждане на списък на компресираните файлове с информация за нивото на компресия

Команда – info, подава се път към архивен файл.

За тази функционалност също използваме класа fileInfo. В сортиран ред се извеждат за всеки файл: неговото име, размер, размер след компресиране, с колко процента е намалял файла след компресиране. Размерът след компресиране се изчислява като разликата на началната позиция и крайната позиция в архива.

* 1. Проверка за повреден архив

Изпълнява се чрез команда check като се подава път до архивен файл.

По-горе споменатата чексума е от полза тук. Тя служи за проверка за промени във файла. При компресирани файлове, и най малката промяна в данните може да бъде катастрофална. Промяната дори на един бит в структурата може да бъде част от размер на дърво, или крайна позиция в метаданни или друго, което може да направи файла невъзможен за разархивиране. Ако по някакъв начин бъдат презаписани, премахнати или добавени данни във файла, то промяната ще бъде или в 4-те байта на чексумата или в останалата част от файла. И в двата случея чексумата и файла вече няма да си съответстват и ще разберем че файлът е бил повреден. Ако пък файла не е архивен файл създаден от нашата програма, то последните 4 байта най-вероятно няма да бъдат чексума генерирана с този алгоритъм и ще разберем че файлът не може да се разархивира.

**CRC\_32 (32-bit Cyclic Redundancy Check) алгоритъм:**

Алгоритъма за обикновена чексума е лесен и се състои в това да събираме числовите стойности на символите във файла в n-битова стойност като позволяваме да overflow-не. Това решение обаче не засича някои промени. Например, ако премахнем символ от едно място и добавим същия символ на друго, чексумата ще остане същата. За това имаме нужда от по-силно и комплексно решение на проблема със засичането на грешки.

Чексумата от типа crc\_32 е един от най-популярните начини за проверка за промяна в данните на файл. По-коректно е да се нарича check-стойност, но ще го наричам чексума за простота. Теоретично, алгоритъмът представлява делене на полиноми от полето на полиномите с коефициенти от булевото поле. Въвеждаме съответните операции събиране, изваждане умножение и деление от булевото поле (което е крайно поле). Знаем че множеството от полиномите от това поле с дължина до 32 е биективно съпоставимо на множеството от 32-битовите числа. Изчисляваме остатъка от делението на полинома образуван от битовете на файла с константен полином от 33-та степен. Резултата е полином от най-много 32-ра степен или 32-битово число. Този остатък наричаме нашата чексума за този файл. Има особености по имлементацията, които са тежки за обяснение, затова ще оставя в материали статията „Understanding CRC32“ от https://commandlinefanatic.com, която четох и която основно повлия моята имплементация на алгоритъма. Метода е известен и е с доказана ефективност и ефикасност.

* 1. Обновяване на вече архивиран файл с нова версия на оригиналния

Случва се чрез команда update като се подава път до обновен файл някъде в системата и път до архивния файл.

Прави се проверка дали файл с такова име съществува в архивния файл. Ако е наличен се сравняват чексумите и размерите на файловете. Ако поне едно не съвпада, трябва да обновим файла. За целта създаваме нов архивен файл. Записваме данните които остават непроменени (от началото до началната позиция на търсения файл в архива) в новия файл. След което архивираме обновения файл и го съхраняваме в новосъздадения архивен файл. Копираме останалата част (от крайната позиция на търсения файл до края) в новия файл. Изтриваме стария файл и преименуваме новия със името на стария.

По принципа на Dirichlet е възможно да има различни файлове с еднакви чексуми, дори да е раядкост, затова не можем да разчитаме много на crc\_32 за проверка за автентичност. Въпреки това за целите на този проект разчитам, че ако два файла имат еднакъв размер, име и чексума то става въпрос за два идентични файла и не правя обновяване в този случай.

1. Използвани ресурси, материали и технологии.
2. Програмиране = ++Алгоритми 5то издание, 10.4.2. Алгоритъм на Хъфман
3. Understanding CRC32 - <https://commandlinefanatic.com/cgi-bin/showarticle.cgi?article=art008>
4. STD filesystem library - <https://en.cppreference.com/w/cpp/filesystem>

Проекта е разработен със:

Компилатор: MSVC

Среда: Visual Studio 2019 (v142)

Операционна система: Windows 10

C++ standart: std::c++latest