

***Проект по избираемия предмет „СДА – Практикум“***

***Алгоритъм на Хъфман***

*Изготвен от: Светлин Попиванов*

1. Описание на използвания алгоритъм

Главният алгоритъм, използван в рамките на този проект, е алгоритъмът на Хъфман. Друг алгоритъм е сериализация/десериализация на бинарно дърво, с цел да се запише дървото на хъфман във отделен файл, така че да бъде извличано при декомпресия. Ще преминем на кратко през тези двата алгоритъма, обяснявайки какво правят те и как го правят те.

Алгоритъмът на Хъфман в своята същност не е толкова сложен за разбиране и реализация. Изискват се основни познания за бинарни дървета и честотни таблици. При постъпване на име на файл за компресия, зададен от потребителя, стринговия низ в него се извлича посредством входно-изходни операции и се изпраща на алгоритъма на Хъфман, който се реализира в рамките на няколко стъпки:

1. Първоначално се създава честотна таблица на низа, тоест на всеки символ от низа ще съответства едно число – неговия брой на срещания в низа. Това се осъществява чрез C++ конструкцията unordered\_map<char, int>, което всъщност е hash map, с целта честотната таблица да бъде построена за линейно време спрямо дължината на низа.
2. За всеки символ от честотната таблица се създава дърво и всяко едно се записва в приоритетна опашка (priority\_queue), в която приоритет се задава върху честотата (frequency променливата на всеки node), тоест най-малката честота ще е винаги на първо място (на върха на heap-a).
3. Следва конструирането на дървото на Хъфман. Взимат се двете дървета с най-малки честоти в корените си от приоритетната опашка и се създава ново дърво, чиято честота е сбора от двете честоти. Лявото поддърво на новосъздаденото става онова дърво, което е изтеглено от опашката първо (тоест с най-малка честота), а дясното поддърво – другото. Новото дърво се вкарва в опашката и тази стъпка се повтаря отново, докато в опашката не остане един елемент – корена на дървото на Хъфман.
4. Следва построяването на азбуката по дървото на Хъфман, тоест отговарящият бинарен код на всеки символ от низа. Това става чрез обхождане в дълбочина (DFS алгоритъм), като всеки код се пази в hash map – unordered\_map<char, string>.

Този алгоритъм, който ползваме, поддържа и низ, който ще представлява кода на всяко едно листо, като при всяко преминаване през ребро се записва ‚0‘ или ‚1‘ в края му, а при преминаване обратно през ребро (преминаване към бащиния node на текущия) се премахва последния записан символ в низа (тоест реброто, през което е преминал). По този начин, когато се стигне до листо в обхождането, в hash map-a се записва акумулирания код на съответния символ, намиращ се в данните на листото.

1. След като вече имаме тази таблица със съответните кодове на всеки символ, можем да компресираме входящият низ. За тази цел, ние просто преминаваме през съответния низ и на мястото на съответния символ записваме неговия код в друг низ. Този друг низ на края ще съдържа компресирания входящ низ, като се добавят нули към края му, ако неговата дължина не се дели модулно по 8 (тоест ако нямаме точен брой байтове). Тези байтове след това се записват в компресиран файл с името (file name)\_compressed.txt

При постъпване на файл за декомпресиране се извлича дървото на Хъфман, посредством десериализация на бинарно дърво и по него се обхожда бинарната последователност, която сме получили от входящия файл. Това обхождане става посредством DFS алгоритъм, който при среща на листо записва съответния символ и се рестартира от корена. По този начин се декомпресира входящата бинарна последователност и резултатният низ (тоест първоначалния) се записва във файл с името (file name)\_decompressed.txt

Алгоритмите за сериализация и десериализация на дървото не са от съществено значение за този проект и затова ще преминем през тях без да се впускаме в подробности. Накратко, сериализацията извършва обхождане в ширина на дървото на Хъфман (BFS) и записва всяко frequency на всеки node, като ако този node е листо, се записва следната последователност: [frequency] -1 -1 [char]. По този начин, десериализиращит алгоритъм ще разпознае листата от другите node-ове. Десериализиращия алгоритъм от друга страна, преминава през този стринг, който сме извели от сериализиращия алгоритъм, като записва всяка честота в опашка под формата на node. Когато срещне листо, той записва съответния символ в последно добавения node в опашката. След това изважда по два node-а от опашката, конструира дърво от тях и вкарва корена на дървото в опашката. Последната стъпка се повтаря, докато не бъде построено дървото на Хъфман.

1. Тестване на алгоритъма

Тестването на този проект се извърши в две фази:

* Първата бе да се провери дали дървото на Хъфман е построено коректно и дали бинарната последователност и числовата (числа от 0 до 255, отговарящи на всеки байт от бинарната последователност) са правилни. Тези тестове могат да бъдат видени в папката tests със следните имена: example{1-4}.txt. Идеята на тези тестове бе да се достигнат определен брой от честоти на всеки символ с цел счупване на кода. Пример за такъв тест е низът „A13aa-bAaB-1B3a-Aaa3b-AA3333--bbaBa---aaabbBBbab--abBaab-BBB-B—„, намиращ се в example2.txt. В него има 15 срещания на символите “-“ и “а”, 10 на “B” и “b”, 7 на “3”, 5 на “А” и 2 на “1”. Както виждаме имаме два символа с най-високи честоти, но въпреки това алгоритъмът правилно построи дървото, като и на двата символа даде код с дължина 2. Относно “B” и “b”, те също получиха еднакъв по дължина код. Конструирането на бинарната последователност, отговаряща на входящия низ, също бе коректна (както и числовата). Те бяха проверени на ръка. Останалите тестове са сходни на примерния, като и те също конструират дървото по правилен начин и по него образуват коректна бинарна последователност (както и числова).
* Втората фаза бе да се избират текстове с произволна дължина и да се тества декомпресията дали връща един и същи низ като първоначалния. Тези тестове преминаха успешно и те се намират във файловете с имена example{5-8}.txt. Пример за такъв тест е например текстът, намиращ се във файла example6.txt. В него се намира един абзац от генериран текст “Lorem ipsum dolor…”. След компресиране и декомпресиране резултатният низ от декомпресията бе проверен в отделна програма дали е един и същ с първоначалния. Това действително бе така. Също така декомпресирания файл бе еднакъв по големина с първоначалния, което означава, че алгоритъма работи коректно.

Важно е да се отебелжи, че и в двете фази на тестване, се проверяваше дали размера на компресирания файл е по-малък по размер от входния. Всеки един такъв файл завършва с \_compressed.txt. Техният размер в действителност е по-малък от съответните им входящи такива. Файловете, които звършват с \_binary.txt и \_decimal.txt са реализирани, за да може да се види бинарната и числовата последователност, която алгоритъма генерира от съответния входящ стринг. Те са особено полезни в първата фаза, където трябва да се провери коректността на „вътрешната“ реализация на алгоритъма (тоест стъпките, през които преминава нашия алгоритъм дали работят коректно).