Huffman coding – Найден Костов Найденов 81944

1. Интродукция

Huffman coding е алгоритъм за компресиране на данни. Идеята е всеки един символ да се записва вместо с 1 байт(8 бита) всеки, с няколко бита в зависимост колко важен е този символ. Това се определя от това колко пъти е срещнат в файла, който ще записваме. Алгоритъмът може да се раздели на 3 стъпки. Създаване на честотна таблица. Създаване на дърво и компресиране или декомпресиране като последна стъпка.

1. Използване

Програмата приема аргументи от конзолата при пускане. Разширенията са примерни. Работи и с други

* 1. След -i filename.txt
  2. След -o outputname.bin
  3. -c Режим компресиране
  4. -d Режим декомпресиране
  5. -debug Режим за дебъгване

Например, за да компресираме файла t.txt във compressed.bin трябва да напишем.

./a.exe -i t.txt -o compressed.bin -c

След това, за да го декомпресираме в нов файл ползваме

./a.exe -i compressed.bin -o new.txt -d

New.txt трябва да съдържа същия текст както в t.txt ако всичко е както трябва. За да видим само резултатите на конзолата е достатъчно да напишем

./a.exe -i t.txt -debug

Това първо ще принтира дървото на Хъфман и след това битовете разделени на по 8.

1. Имплементация
   1. Създаване на честотна таблица – Използвам hash map за съдържането на данните. За всеки нов символ се мапва един инт, който се увеличава при всяко ново срещане. Избрах unordered\_map от stl, защото търсенето ще е средно константно време.
   2. Създаване на дърво на Хъфман – Тук ползвам допълнителна структура от данни PriorityQueue, която я има в stl, но съм направил и моя имплементация която е малко по различна.
2. Първата стъпка е да попълним приоритетната опашка с елементите от честотната таблица като листа.
3. Самото дърво се строи отдолу нагоре. Вземат се двете най-малки под дървета и след това се прави нов Node, който има за приоритет сбора на приоритетите на левия и десния си наследник.
4. Повтаряме стъпка 2 докато не останем само с 1 Node и той ни е root
5. Допълнителна стъпка за запазване на кода на всеки символ в hash map lookup, за да имаме достъп директно когато компресираме по късно. Понеже кода е бинарен и може и да е повече от 8 бита. Поради това съм направил нов клас за работа с бинарни операции BinaryBuffer, който се грижи за това.
   1. Компресиране – Първо записваме в хедъра дървото на Хъфман. Обхождаме го с DFS и използвам един специален символ ~ за да кажа че тук има Node и някакъв друг символ когато е необходимо. След това се минава символ по символ и търсим в lookup и го добавяме в един общ BinaryBuffer. Ако е в режим debug просто принтираме този буфер на конзолата. Иначе го записваме във output файла.
   2. Декомпресиране – Четем хедъра с рекурсивна функция и попълваме дървото на Хъфман. След това вземаме байт по байт записаните данни във файла и обхождаме дървото бит по бит. Бит 0 отиваме наляво бит 1 надясно и когато стигнем листо го записваме в output файла и почваме отново от root да обхождаме
6. Допълнителни класове
   1. BinaryBuffer (Най – объркващия клас, който съм писал)

Самият буфер представлява вектор от char или байтове и се орпавя с подравняване и други проблеми. Помощните функции shiftLeft и shiftRight преместват буфера с едно наляво или едно надясно като имаме една променлива index, която показва до колко ни е запълнен текущият бит. shiftLeft шифтва 1 наляво и променяме индекса на последния елемент с +1. Ако ни е запълнен добавяме нов бит и извършваме операцията. По подобен начин с shiftRight шифтва 1 надясно и всъшност се премахва -1 от индекса. Другите функции се свеждат до тези 2 основни. Също така и оператор ++, който променя последния бит с 1.

* 1. PriorityQueue (Още по объркващ клас, бонус)

Реших да не го направя с вектор както всички други имплементации. Направих го с Node, който има left, right, bool side, T data. Единствената разлика е булевата променлива в Node. Тя всъшност прави обхождането до листо. Идеята е че първо минаваме на ляво и при второ преминаване отиваме надясно. Тоест когато ни дадът 1 2 3. 1 Става root, 2 отива в ляво и 3 в дясно и при следващото вмъкване ще започнем от лявото дете на 2 и следващото лявото дете на 3. Когато дойде нов елемент минавам по дървото докато този елемент е по-малък от текущия и когато се наруши това условие текущия става новия елемент, а стария текущ продължаваме да го вмъкваме надолу в поддървото докато не стигнем листо. Слъжността е O(log n) винаги, така че не е по-бързо от обикновен binary heap, но целия клас го направих за упражнение. Pop премахва първия елемент и го замества с по-малкия му наследник. След това прави същото на ляво или надясно в зависимост от наследника, докато не стигне листо в който случай го изтрива. Отново O(log n) сложност.

5.Резултати

Lorem ipsum текст от 3 kb се компресира до 1.65 kb с моята имплементация на PriorityQueue и същото с stl priority queue. В zip формат се компресира до 1.33 kb. Може би защото записвам самото дърво символ по символ, а не вместо този допълнителен символ ~ да се ползва 1 и също цялото дърво да се запише побитово. 37 ~ по 8 бита всяка ако ги направя по 1 бит щях да получа точно същото ниво на компресиране както zip.