**Національний технічний університет України «КПІ»**

**Інститут прикладного системного аналізу**

**Кафедра Системного проектування**

Лабораторна рoбота

З теорії інформації та кодування

«Алгоритм Хаффмана. Архіватор. Деархіватор»

Виконав:

студент 2 курсу

групи ДА-21

Круша Ігоря

Київ – 2013

Алгоритм Хаффмана

Алгоритм Хаффмана - адаптивний жадібний алгоритм оптимального префіксних кодування алфавіту з мінімальною надмірністю. Був розроблений в 1952 році аспірантом Массачусетського технологічного інституту Девідом Хаффманом при написанні їм курсової роботи. В даний час використовується в багатьох програмах стиснення даних.

На відміну від алгоритму Шеннона - Фано, алгоритм Хаффмана залишається завжди оптимальним і для вторинних алфавітів m2 з більш ніж двома символами.

Цей метод кодування складається з двох основних етапів:

1.Побудова оптимального кодового дерева.

2.Побудова відображення код-символ на основі побудованого дерева.

Розлянемо цей алгоритм на основі прикладу:

Припустимо, що у нас є строка «beep boop beer!», для якої, в її поточному вигляді, на кожен знак витрачається по одному байту. Це означає, що вся рядок цілком займає 15 \* 8 = 120 біт пам'яті.

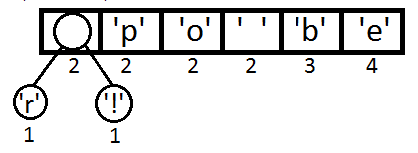
Для початку побудуємо таблицю частот цих символів:

|  |  |
| --- | --- |
| **Символ** | **Частота** |
| 'b' | 3 |
| 'e' | 4 |
| 'p' | 2 |
| ' ' | 2 |
| 'o' | 2 |
| 'r' | 1 |
| '!' | 1 |

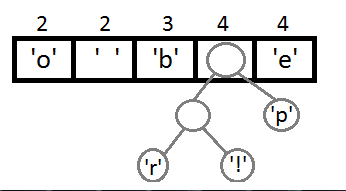
Після обчислення частот ми створимо вузли бінарного дерева для кожного знака і додамо їх в чергу, використовуючи частоту в якості пріоритету:

A description...

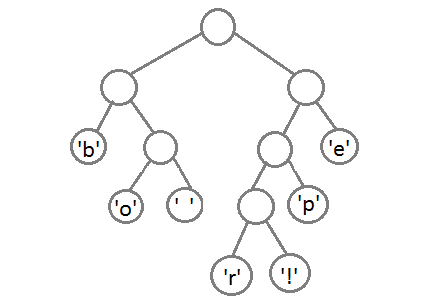
Тепер ми дістаємо два перших елемента з черги і пов'язуємо їх, створюючи новий вузол дерева, в якому вони обидва будуть нащадками, а пріоритет нового вузла буде дорівнює сумі їх пріоритетів. Після цього ми додамо вийшов новий вузол назад у чергу.



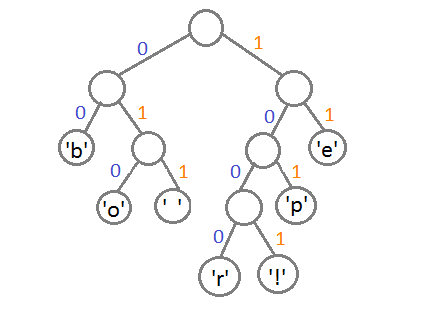
Повторимо ті ж кроки і отримаємо:



Використовуючи вищенаведені правила прийдемо до такого вигляду дерева.



Тепер, щоб отримати код для кожного символу, треба просто пройтися по дереву, і для кожного переходу додавати 0, якщо ми йдемо вліво, і 1 - якщо направо:



Ми отримали нові коди символів для даної строки:

|  |  |
| --- | --- |
| **Символ** | **Код** |
| 'b' | 00 |
| 'e' | 11 |
| 'p' | 101 |
| ' ' | 011 |
| 'o' | 010 |
| 'r' | 1000 |
| '!' | 1001 |

Щоб розшифрувати закодований рядок, нам треба, відповідно, просто йти по дереву, згортаючи у відповідну кожному біту сторону до тих пір, поки ми не досягнемо листа. Наприклад, якщо є рядок «101 11101 11» і наше дерево, то ми отримаємо рядок «pepe».

Важливо мати на увазі, що кожен код не є префіксом для коду іншого символу. У нашому прикладі, якщо 00 - це код для 'b', то 000 не може виявитися чиєюсь кодом, тому що інакше ми отримаємо конфлікт. Ми ніколи не досягли б цього символу в дереві, так як зупинялися б ще на 'b'.

Архіватор та деархіватор було реалізовано в одній програмі.

**Архіватор**

Зчитуємо файл побітово в асоціативний масив. На його базі будуємо дерево. Після цього будуємо таблицю(асоціативний масив), в якій перша колонка — символи, друга — масив з нулів та одиниць(таблиця при цьому записується у файл). Конвертуємо дерево в строку, заносимо її у файл. Далі побітово записуємо коди символів на їх місці.

**Деархіватор**

Відкриваємо файл, за допомогою рекурсивної функції будуємо дерево. Далі зчитуємо побітово і в дереві шукаємо символ(так як алгоритм Хаффмана непрефіксний, кожен символ має свій код, тож розкодовується однозначно).

Код:

**#include <iostream>**

**#include <cstdlib>**

**#include <vector>**

**#include <map>**

**#include <list>**

**#include <fstream>**

**#include <ctime>**

**using** **namespace** std;

*/\*\**

*\* class Node*

*\* for tree*

*\*/*

**class** **Node** {

public:

**int** a;

**char** c;

Node **\***left, **\***right;

Node() {

left **=** right **=** NULL;

}

Node(Node **\*** l, Node **\*** r) {

a **=** l**->**a **+** r**->**a;

left **=** l;

right **=** r;

}

};

*/\*\**

*\* class myCompare*

*\* class for comparing*

*\*/*

**class** **myCompare** {

public:

**bool** **operator**()(Node **\*** l, Node **\*** r) **const** {

**return** l**->**a **<** r**->**a;

}

};

*/\*\**

*\* test function for printing tree*

*\*/*

**void** **print**(Node **\*** root, **unsigned** **int** k **=** 0) {

**if**(root **!=** NULL) {

print(root**->**left, k**+**3);

**for**(**unsigned** **int** i **=** 0; i **<** k; i**++**) {

cout **<<** " ";

}

**if**(root**->**c) {

cout **<<** root**->**a **<<** " (" **<<** root**->**c **<<** ") " **<<** endl;

} **else** {

cout **<<** root**->**a **<<** endl;

}

print(root**->**right, k**+**3);

}

}

*// global variables*

*// code - vector for each symbol's code*

vector**<bool>** code;

map**<char**, vector**<bool>** **>** table;

*/\*\**

*\* function buildTable*

*\* recursive function for building associative array - table*

*\*/*

**void** **buildTable**(Node **\***root, ofstream **&**f) {

**if**(root**->**left **!=** NULL) {

code.push\_back(0);

buildTable(root**->**left, f);

}

**if**(root**->**right **!=** NULL) {

code.push\_back(1);

buildTable(root**->**right, f);

}

**if**(root**->**c) {

**for**(**int** i **=** 0; i **<** code.size(); i**++**) {

f **<<** code[i];

}

f **<<** " " **<<** root**->**c **<<** '\n';

table[root**->**c] **=** code;

}

code.pop\_back();

}

*/\*\**

*\* function convertFromTree*

*\* convert from tree to file*

*\*/*

**void** **convertFromTree**(Node **\*** root, ofstream **&**f) {

**if**(root**->**left) {

**if**(root**->**left**->**c) {

f **<<** 0 **<<** root**->**left**->**c;

} **else** {

f **<<** 1;

}

convertFromTree(root**->**left, f);

}

**if**(root**->**right) {

**if**(root**->**right**->**c) {

f **<<** 0 **<<** root**->**right**->**c;

} **else** {

f **<<** 1;

}

convertFromTree(root**->**right, f);

}

}

*/\*\**

*\* function convertToTree*

*\* convert form file to tree*

*\*/*

**void** **convertToTree**(Node **\*\*** root, ifstream **&**f) {

**int** c **=** f.get();

**if**(c **==** 49) {

(**\***root)**->**left **=** **new** Node();

(**\***root)**->**right **=** **new** Node();

convertToTree(**&**((**\***root)**->**left), f);

convertToTree(**&**((**\***root)**->**right), f);

} **else** **if**(c **==** 48) {

**char** sym **=** f.get();

(**\***root)**->**c **=** sym;

}

}

**int** **main**() {

**bool** exit **=** false;

srand(time(NULL));

**while**(**!**exit) {

*// getting a file type*

**int** type;

cout **<<** "If you want to encode, put 0; decode - put 1" **<<** endl;

cin **>>** type;

cin.ignore(1);

*// getting files*

string input, output;

cout **<<** "Please, input in filename" **<<** endl;

cin **>>** input;

cin.ignore();

cout **<<** "Please, input out filename" **<<** endl;

cin **>>** output;

map**<char**, **int>** chars;

*// for measuring of time*

**clock\_t** progTime **=** clock();

*// type == 0 - encode*

**if**(type **==** 0) {

ifstream f(input.c\_str(), ios**::**binary);

**if**(f.fail()) {

cout **<<** "File is not exist" **<<** endl;

**return** 0;

}

*// getting the chars*

**while**(**!**f.eof()) {

**char** c **=** f.get();

chars[c]**++**;

}

map**<char**, **int>::**iterator itr;

list**<**Node**\*>** tree;

*// creating of the tree*

**for**(itr **=** chars.begin(); itr **!=** chars.end(); **++**itr) {

Node **\***p **=** **new** Node();

p**->**c **=** itr**->**first;

p**->**a **=** itr**->**second;

tree.push\_back(p);

}

**while**(tree.size() **!=** 1) {

tree.sort(myCompare());

Node **\***sonL **=** tree.front();

tree.pop\_front();

Node **\***sonR **=** tree.front();

tree.pop\_front();

Node **\***parent **=** **new** Node(sonL, sonR);

tree.push\_back(parent);

}

Node **\*** root **=** tree.front();

*// creatinf o statistic file*

string outputStatistic **=** output **+** "\_statistic.txt";

ofstream outStat(outputStatistic.c\_str(), ios**::**out **|** ios**::**binary);

*// building of a table*

buildTable(root, outStat);

f.clear();

f.seekg(0);

ofstream out((output **+** ".bin").c\_str(), ios**::**out **|** ios**::**binary);

*// writing of tree into the file*

out **<<** 1;

convertFromTree(root, out);

*// writing the code to the file*

**int** count **=** 0; **char** buf **=** 0;

**while**(**!**f.eof()) {

**char** c **=** f.get();

vector**<bool>** x **=** table[c];

**for**(**int** n **=** 0; n **<** x.size(); n**++**) {

buf **=** buf **|** x[n] **<<** (7 **-** count);

**if**(**++**count **==** 8) {

count **=** 0;

out **<<** buf;

buf **=** 0;

}

}

}

f.close();

out.close();

} **else** { *// type == 1 - decode*

ifstream in(input.c\_str(), ios**::**in **|** ios**::**binary);

ofstream out(output.c\_str(), ios**::**out **|** ios**::**binary);

Node **\*** treeFromFile **=** **new** Node();

*// building a tree from the file*

convertToTree(**&**treeFromFile, in);

**int** count **=** 0;

**char** byte;

Node **\***p **=** treeFromFile;

*// decoding*

byte **=** in.get();

**while**(**!**in.eof()) {

**bool** b **=** byte **&** (1 **<<** (7 **-** count));

**if**(b) {

p **=** p**->**right;

} **else** {

p **=** p**->**left;

}

**if**(p**->**left **==** NULL **&&** p**->**right **==** NULL) {

out **<<** p**->**c;

p **=** treeFromFile;

}

count**++**;

**if**(count **==** 8) {

byte **=** in.get();

count **=** 0;

}

}

in.close();

out.close();

treeFromFile **=** NULL;

}

progTime **=** clock() **-** progTime;

cout **<<** "Time: " **<<**(**double**)progTime **/** CLOCKS\_PER\_SEC **<<** " seconds" **<<** endl;

code.clear();

table.clear();

cout **<<** "Exit? 1/0" **<<** endl;

cin **>>** exit;

}

**return** 0;

}

Висновок.

Реалізований архіватор по методу Хаффмана та деархіватор. Побудовані тестові приклади, що характеризують продуктивність rar, zip та даного архіваторів.