

Курсовая работа по курсу дискретного анализа: Архиватор Хаффмана

Выполнил студент группы М80-308Б-20 Морозов Артем Борисович.

Условие

Необходимо реализовать два известных метода сжатия данных для сжатия одного файла.

Методы сжатия выбираются из следующих групп:

- Арифметическое кодирование, кодирование по Хаффману
- LZ77, LZW, BWT+MTF+RLE

Формат запуска должен быть аналогичен формату запуска программы gzip. Должны быть поддерживаться следующие ключи: -c, -d, -k, -l, -r, -t, -1, -9. Должно поддерживаться указание символа дефиса в качестве стандартного ввода.

Метод решения

К программе, реализующей LZ77, в данной вариации курсового проекта добавился так же алгоритм Хаффмана для кодирования. Я выбрал полустатистический алгоритм Хаффмана – когда мы не динамически расширяем дерево, а сначала делаем полный препроцессинг (считаем частоту каждого символа в тексте, создаем ноды с символами и строим полностью дерево Хаффмана по отсортированному вектору наших вершин с символами, где на нижних уровнях у нас будут символы с маленькими частотами, а на верхних – с большими).

Алгоритм Хаффмана устроен так, что полученная битовая последовательность для каждого символа соответствует только ему, в связи с чем полученная кодировка уникальна и правильна.

Сложность кодирования при таком подходе – $O(n)$, где n – количество букв в тексте, так как мы никогда не делаем более чем n операций (формируем массив частот, проходимся по массиву вершин).

Сложность декодирования при таком подходе также $O(n)$, где n – количество букв в тексте, так как мы тоже никогда не делаем более чем n операций. Помимо этого, были также реализованы флаги, отвечающие за инструкцию по применению, выбор алгоритма, разархивирование, получение информации и вывод в стандартный поток ввода вместо файла.

Описание программы

```
#include <iostream>
#include <map>
#include <fstream>
#include <iostream>
#include <algorithm>
#include <vector>
const char END_OF_FILE = '!';
const char left_way = '0';
const char right_way = '1';
std:: string FILE_NAME;
using namespace std;
//<offset, size, next> - code
struct Node {
    int offset;
    int size;
    char next;
    Node (int offset, int size, char next) {
        this->offset = offset;
        this->size = size;
        this->next = next;
    }
};
void LZ77Encode (std:: vector<char>& text, std:: vector<Node*>& encoded) {
    int global_position = 0;
    while (global_position < text.size()) {
        Node* result = new Node(0, 0, text[global_position]);
        for (int i = 0; i < global_position; ++i) {
            int local_position = global_position;
            int local_size = 0;
            for (int j = i; text[local_position] == text[j] && local_position < text.size(); ++j) { //смещаемся по тексту и ищем самый большой size
                ++local_size;
                ++local_position;
            }
            if (local_size >= result->size && local_size != 0) {
                result->size = local_size;
                result->offset = global_position - i;
                if (local_position >= text.size()) { //если превысили или на конце текста
                    result->next = END_OF_FILE;
                }
                else {
                    result->next = text[local_position]; //кладем следующий символ, если
                    все ок
                }
            }
        }
        encoded.emplace_back(result);
        global_position += result->size + 1; // смещаем наш итератор на следующую ячейку
        после последнего обработанного символа
    }
}
void LZ77EncodeWithFile(std:: string& file, bool c, bool l) { //WORKS
    std:: ifstream f(file);
    std:: vector<Node*> encoded;
    std:: vector<char> word = std:: vector<char>(std:: istreambuf_iterator<char>(f),
```

```

std:: istreambuf_iterator<char>());
f.close();
LZ77Encode(word, encoded);
if (l) { //works
    std:: cout << "Result of LZ77 encoding algorithm for file " << file << "\n";
    std:: cout << "Size of original word: " << word.size() * 8 << " bytes\n"; //1byte
= 8bits
    std:: cout << "Size of compressed text: " << encoded.size() * 8 << " bytes\n";
}
if (!c) { //works
    std:: string name = file + ".LZ77";
    std:: ofstream f(name);
    for (int i = 0; i < encoded.size(); ++i) {
        f << encoded[i]->offset << " " << encoded[i]->size << " " << encoded[i]->next
<< "\n";
    }
    f.close();
}
else { //works
    for (int i = 0; i < encoded.size(); ++i) {
        std:: cout << encoded[i]->offset << " " << encoded[i]->size << " " <<
encoded[i]->next << "\n";
    }
}
}
void LZ77Decode (std:: string& text, std:: vector<Node*>& code) {
    int global_position = 0;
    for (int i = 0; i < code.size(); ++i) {
        if (code[i]->size > 0) { //если есть что брать
            global_position = text.size() - code[i]->offset; //смещаемся на длину строки
- смещение
            for (int j = 0; j < code[i]->size; ++j) {
                text.push_back(text[global_position + j]); //прибавляем в текст пока j <
размера того, что берем
            }
        }
        if (code[i]->next == END_OF_FILE) { //если пустой, выходим
            break;
        }
        else {
            text.push_back(code[i]->next); //добавляем следующий символ в текст, если не
пустой символ
        }
    }
}
void LZ77DecodeWithFile(std:: string& file, bool c, bool l) { //WORKS
    std:: ifstream f(file);
    std:: vector<Node*> code;
    int offset, size;
    char next;
    while (f >> offset >> size >> next) {
        Node* inputed = new Node(offset, size, next);
        code.emplace_back(inputed);
    }
    f.close();
    std:: string text;
    LZ77Decode(text, code);
    std:: string original_filename = file.substr(0, file.size() - 5); //test.txt
}

```

```

    if (l) {
        std:: cout << "Result of LZ77 decoding algorithm for file " << original_filename
<< "\n";
        std:: cout << "Size of original word: " << text.size() * 8 << " bytes\n";
        std:: cout << "Size of compressed text: " << code.size() * 8 << " bytes\n";
    }
    if (!c) {
        std:: ofstream f("decoded" + original_filename);
        f << text;
        f.close();
    }
    else {
        std:: cout << text << "\n";
    }
}

//HUFFMAN PART
struct HuffmanNode {
    HuffmanNode *left;
    HuffmanNode *right;
    char symbol;
    int frequency;
    HuffmanNode (HuffmanNode* left, HuffmanNode* right, char symbol, int frequency) {
        this->left = left;
        this->right = right;
        this->symbol = symbol;
        this->frequency = frequency;
    }
    ~HuffmanNode() {}
};

//char to binary string
std:: string MakeBinaryString (const char& symbol){
    char str[9] = {0};
    for (int i = 8; i--;) {
        str[7 - i] = !(symbol & (1 << i)) + '0';
    }
    std:: string binarystring = str;
    return binarystring;
}
// binary string to char
char MakeCharFromBinaryString(const std:: string& word) {
    char res = 0;
    int temp = 1;
    for (int i = 7 ; i >= 0; --i) {
        if (word.at(i) == '1') {
            res += temp;
        }
        temp = temp * 2;
    }
    return res;
}
void MakeFrequencyArray(std:: vector<int>& frequency_array, const std:: vector<char>& word) { //массив частот для каждого символа works
    for (int i = 0; i < word.size(); ++i) { //заполняем частоты
        int current_char = (int)word[i];
        ++frequency_array[current_char];
    }
}

```

```

}

struct {
    bool operator() (HuffmanNode *first, HuffmanNode *second) const { return first->frequency > second->frequency; }
} customLess;
void FillNodes(std:: vector<HuffmanNode*>& nodes, std:: vector<int>& frequency_array) {
//WORKS
    for (int i = 0; i < frequency_array.size(); ++i) {
        if (frequency_array[i]) {
            //create new node
            char node_symbol = (char)i;
            HuffmanNode* node = new HuffmanNode(nullptr, nullptr, node_symbol,
frequency_array[i]);
            nodes.emplace_back(node);
        }
    }
}
HuffmanNode* MakeHuffmanTree(std:: vector<HuffmanNode*>& nodes) {
    std:: sort(nodes.begin(), nodes.end(), customLess);
    while (nodes.size() != 1) {
        std:: sort(nodes.begin(), nodes.end(), customLess);
        HuffmanNode* first = nodes.back();
        nodes.pop_back();
        HuffmanNode* second = nodes.back();
        nodes.pop_back();
        int merged_frequency = first->frequency + second->frequency;
        HuffmanNode* merged_node = new HuffmanNode(first, second, '\0',
merged_frequency);
        nodes.emplace_back(merged_node);
    }
    HuffmanNode* result = nodes.back();
    return result;
}
void MakeBinaryCode (HuffmanNode* node, std:: string code, std:: map<char, std:: string>& codes) {
    if (node == nullptr) {
        return;
    }
    if (!node->left && !node->right) {
        if (code.empty()) {
            code = "0";
        }
        codes[node->symbol] = code;
    }
    MakeBinaryCode(node->left, code + left_way, codes);
    MakeBinaryCode(node->right, code + right_way, codes);
}
void HuffmanEncodeWithFile(std:: string &name, bool c, bool l) {
    std:: ifstream file(name);
    std:: vector<char> word = std:: vector<char>(std:: istreambuf_iterator<char>(file),
std:: istreambuf_iterator<char>());
    int word_size = word.size();
    file.close();
    std:: vector<int> frequency_array(256, 0); //массив частот на 256 символов по
умолчанию 0
    std:: vector<HuffmanNode*> nodes; //массив с нодами
    MakeFrequencyArray(frequency_array, word); //заполняем массив частот
    FillNodes(nodes, frequency_array); //формируем ноды для каждой встречающейся буквы
}

```

```

HuffmanNode* tree = MakeHuffmanTree(nodes); //получаем дерево хафмана
std::map<char, std::string> codes;
std::string begin = "";
MakeBinaryCode(tree, begin, codes); //обход дерева чтобы получить бинарный код
каждого чара
std::string vocabulary_name = name + ".HuffmanVocabulary";
std::ofstream f;
f.open(vocabulary_name);
f << word_size << "\n";
for (std::map<char, std::string>::iterator it = codes.begin(); it != codes.end();
++it) {
    f << it->first << " " << it->second << "\n";
}
f.close();
std::string result = "";
std::string result_name = name + ".Huffman";
std::ofstream Huffman;
Huffman.open(result_name, std::ios::out | std::ios::binary | std::ios::app);
for (int i = 0; i < word.size(); ++i) {
    result += codes[word[i]]; //итоговая строка из 0 и 1 на последней итерации
    while (result.size() >= 8) {
        std::string to_code = result.substr(0, 8); //читываем по восьмеркам в чар
        result.erase(0, 8);
        char coded = MakeCharFromBinaryString(to_code); //создаем чар и пишем в
бинарник
        if (!c) {
            Huffman.write(reinterpret_cast<char*>(&coded), sizeof(coded));
        }
        else {
            std::cout << coded;
        }
    }
}
if (result.size()) {
    result += "0000000";
    std::string to_code = result.substr(0, 8);
    char coded = MakeCharFromBinaryString(to_code);
    if (!c) {
        Huffman.write(reinterpret_cast<char*>(&coded), sizeof(coded));
    }
    else {
        std::cout << coded << "\n";
    }
}
Huffman.close();
if (1) {
    std::ifstream output(result_name, std::ios::binary | std::ios::ate);
    int output_size = output.tellg();
    output.close();
    std::cout << "Result of Huffman encoding algorithm for file " << name << "\n";
    std::cout << "Size of original word: " << word.size() * 8 << " bytes\n";
    std::cout << "Size of compressed text: " << output_size * 8 << " bytes\n";
}
}

void HuffmanDecompressWithFile(std::string &name, bool c, bool l) {
    int word_size;
    std::string str;

```

```

char ch;
std:: ifstream vocabulary_read;
vocabulary_read.open(name + "Vocabulary");
vocabulary_read >> word_size;
std:: map<std:: string, char> codes;
while (vocabulary_read >> ch >> str) {
    codes[str] = ch;
}
std:: ifstream read_huffman;
read_huffman.open(name, std:: ios:: binary | std:: ios:: in);
int decoded_size = 0;
int start_iter;
char current_byte;
int finish_iter;
std:: string coded_string = "";
std:: ofstream res;
res.open("decoded" + name, std:: ios:: binary | std:: ios:: out);
while (read_huffman.read(reinterpret_cast<char*>(&current_byte),
sizeof(current_byte)) && decoded_size < word_size) {
    coded_string += MakeBinaryString(current_byte);
    start_iter = 0;
    finish_iter = 1;
    while (start_iter + finish_iter <= coded_string.size() && decoded_size <
word_size) {
        std:: string potential_code = coded_string.substr(start_iter, finish_iter);
        std:: map<std:: string, char>::iterator iter = codes.find(potential_code);
        if (iter != codes.end()) {
            coded_string.erase(0, start_iter + finish_iter); //удаляем совпадение
            ++decoded_size;
            char symbol = iter->second;
            if (!c) {
                res.write(reinterpret_cast<char*>(&symbol), sizeof(symbol));
            }
            else {
                std:: cout << symbol;
            }
            finish_iter = 1;
        }
        else {
            ++finish_iter;
        }
    }
}
if (l) {
    std:: ifstream output(name, std:: ios:: binary | std:: ios:: ate);
    int output_size = output.tellg();
    std:: cout << "Result of Huffman decoding algorithm for file " << name << "\n";
    std:: cout << "Size of compressed text: " << output_size * 8 << " bytes\n";
    std:: cout << "Size of original word: " << word_size * 8 << " bytes\n";
}
}
void Help() {
    std:: cout << "Hello!" << "\n";
    std:: cout << "Here is instruction for flags: " << "\n";
    std:: cout << "\t" << "-9 for LZ77 coding" << "\n";
    std:: cout << "\t" << "-1 for Huffman coding" << "\n";
    std:: cout << "\t" << "-1 for information about compressed/decompressed and original

```

```

sizes" << "\n";
    std::cout << "\t" << "-c for output in standart thread instead of input in file" <<
"\n";
    std::cout << "\t" << "-d for decoding" << "\n";
    std::cout << "\t" << "-k for information" << "\n";
}
void RunWithFlags(std::string& file, bool lz77, bool huffman, bool c, bool d, bool l) {
    if (lz77) {
        if (d) { //разархивировать при помощи LZ77 {
            LZ77DecodeWithFile(file, c, l);
        }
        else { //заархивировать при помощи LZ77
            LZ77EncodeWithFile(file, c, l);
        }
    }
    else if (huffman) {
        if (d) { //разархивировать при помощи хаффмана
            HuffmanDecompressWithFile(file, c, l);
        }
        else { //заархивировать при помощи хаффмана
            HuffmanEncodeWithFile(file, c, l);
        }
    }
}
int main (int args, char** argv) {
    //инициализируем булевые значения ложью по умолчанию
    bool lz77 = false; // -9 - сжатие при помощи алгоритма LZ-77
    bool huffman = false; // -1 - сжатие при помощи алгоритма Хаффмана
    bool l = false; // информация о сжатом и разжатом размерах
    bool c = false; // false - вывод в файл, true - в окно ввода
    bool d = false; // разархивация
    bool k = false; // информация
    for (int i = 1; i < args; ++i) { // с единички чтобы запуск программы не был как файл
для сжатия
        std::string current = argv[i]; // чтобы не было ошибок сравнения char с const
char*
        if (current == "-9") {
            lz77 = true;
        }
        else if (current == "-1") {
            huffman = true;
        }
        else if (current == "-l") {
            l = true;
        }
        else if (current == "-c") {
            c = true;
        }
        else if (current == "-d") {
            d = true;
        }
        else if (current == "-k") {
            k = true;
        }
        else {
            FILE_NAME = current;
        }
    }
}

```

```

    }
    if (lz77 == false && huffman == false) { //если флагов на сжатие не было введено, по
умолчанию кодируем через lz77
        lz77 = true;
    }
    else if (lz77 == true && huffman == true) { //если оба введены
        std::cout << "You should enter only one key: -1 for huffman coding or -9 for
LZ77 coding!" << "\n";
        exit(EXIT_FAILURE);
    }
    else if (lz77 == false && huffman == false && l == false && c == false && d == false
&& k == false) { //если ничего не введено
        std::cout << "You should enter one more flag. You've entered nothing. Here is
info: " << "\n";
        Help();
        exit(EXIT_FAILURE);
    }
    else if (k) { //информация
        lz77 = false;
        huffman = false;
        l = false;
        c = false;
        d = false;
        Help();
    }
    else {
        RunWithFlags(FILE_NAME, lz77, huffman, c, d, l);
    }
    return 0;
}

```

Дневник отладки

В отладке нуждались как функция кодировки, так и функция декодировки. Были проблемы с переводом битовой строки в символ и обратно, в связи с чем изначально программа работала некорректно. Была так же исправлена проблема с кодировкой, когда в строке оставалось меньше 8 чисел 0 и 1, чтобы перевести их в char, а так же корректный перевод из бинарного вида в изначальный в функции декодировки.

Тест производительности

Сравним алгоритм LZ77 с алгоритмом Хаффмана:

В прошлом бенчмарке тесте получились такие результаты для функции сжатия LZ-77:

1) N = 5000:

21 ms

2) N = 10000:

84 ms

3) N = 20000:

220 ms

4) N = 50000:

1162 ms

5) N = 100000:

4398 ms

И соответственно такие результаты для функции декодирования LZ-77:

1) Декодирование слова в 5000 букв:

5 ms

2) Декодирование слова в 10000 букв:

14 ms

3) Декодирование слова в 20000 букв:

37 ms

4) Закодированное слово в 50000 слов:

104 ms

5) Закодированное слово в 100000 слов:

1094 ms

На аналогичных тестах по количеству букв проверим работу алгоритма Хаффмана:

Функция сжатия:

1) N = 5000

Time: 3ms

2) N = 10000

Time: 5ms

3) N = 20000

Time: 6ms

4) N = 50000

Time: 14ms

5) N = 100000

Time: 29ms

Функция декодирования:

1) N = 5000

Time: 6ms

2) N = 10000

Time: 11ms

3) N = 20000

Time: 22ms

4) N = 50000

Time: 77ms

5) N = 100000

Time: 115ms

Как мы видим, алгоритм LZ77 уступает алгоритму Хаффмана в декодировании лишь на самом большом тесте, а вот в сжатии уступает абсолютно везде и с очень большим отрывом. Оно и неудивительно, ведь алгоритм Хаффмана сжимает за, по сути, линейное время, а LZ77 за кубическое.

Недочёты

Недочетов в программе обнаружено не было, однако стоит упомянуть, что программа работает только при условии корректного ввода, так как была разработана исключительно в учебных целях. Любой неправильный ввод может убить работоспособность моей программы.

Пример работы

Файл:

Сжатие:

```
PS C:\it master\MAI\2 course\5sem\Дискретный анализ\cp5> g++ cp.cpp -o main
PS C:\it master\MAI\2 course\5sem\Дискретный анализ\cp5> ./main -1 -l test.txt
Result of Huffman encoding algorithm for file test.txt
Size of original word: 2224 bytes
Size of compressed text: 904 bytes
```

≡ test.txt.HuffmanVocabulary

1		278
2	A	11100
3	B	110101
4	C	11010001
5	D	10
6	E	11011
7	F	00
8	H	010
9	J	0111
10	K	1100
11	L	11111
12	N	11010000
13	Q	0110
14	R	1101001
15	S	11110
16	W	11101

```
cp.cpp test.txt.Huffman x
```

Декодирование:

```
PS C:\it\master\MAI\2 course\5sem\Дискретный анализ\cp5> ./main -1 -l -d test.txt.Huffman
Result of Huffman decoding algorithm for file test.txt.Huffman
Size of compressed text: 904 bytes
Size of original word: 2224 bytes
```

Выводы

Сделав усложненный курсовой проект по дискретному анализу, я познакомился с таким интересным алгоритмом, как архиватор Хаффмана и научился его программировать.

Повторюсь, что был разработан полустатический вариант алгоритма, так как он проще в понимании и приятнее.

Динамический алгоритм программировать было бы гораздо сложнее.