Московский авиационный институт
(национальный исследовательский университет)
Институт № 8 «Информационные технологии и прикладная математика»

Курсовой проект
по курсу «Дискретный анализ»
«Архиватор: кодирование по Хаффману и алгоритм LZW»

Студент: Минеева Светлана Алексеевна

Группа: М8О-310Б-21

Преподаватель: Сорокин С.А.

Оценка: ____

Дата: 08.01.2024

Подпись: _____

Москва, 2024

Содержание:

1. Постановка задачи

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

- Кодирование по Хаффману
- Алгоритм LZW

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

2. Общий метод и алгоритм решения

LZW (Lempel-Ziv-Welch) - это алгоритм сжатия данных без потерь на основе словаря, который был разработан в 1984 году. Он предназначен для уменьшения объема данных, представляя повторяющиеся последовательности битов более компактно. Особенность и преимущество алгоритма – легкая реализация.

Рассмотрим процесс сжатия данных. Инициализируется словарь, в него помещается алфавит рассматриваемого языка. С входного потока считывается строка символов и посимвольным проходом происходит проверка, существует ли в таблице такая строка. Если есть, то считываем следующий символ, иначе добавляем код для предыдущей найденной строки в результат, строку заносим в таблицу и продолжаем поиск до окончания входного текста. Данный процесс можно визуализировать следующей последовательностью шагов:

Шаг 1. Символы алфавита рассматриваемого языка заносим в словарь. Входная строка A пополняется первым символом текста.

Шаг 2. Считываем следующий символ текста (вид анализируемой строки – AB).

Шаг 3. Если считанный только что считанный символ – это окончание текста, то в результат добавляется код для A, иначе если строка AB есть в словаре, то A = AB и далее шаг 2, а если AB нет в словаре, то в результат добавляем код для строки A, строку AB добавляем в словарь, A = B и далее шаг 2.

Декодирование происходит аналогичным способом. При получении нового кода создается новая строка в словаре, а при получении существующего кода в результат из словаря добавляется строка с рассматриваемым кодом.

Алгоритм Хаффмана — это алгоритм сжатия данных, разработанный американским математиком Дэвидом Хаффманом в 1952 году.

Основная идея алгоритма состоит в том, чтобы присвоить переменную длину кодам символов в соответствии с их частотами встречаемости. Символы, которые часто встречаются, получают короткие коды, а символы, которые редко встречаются, получают более длинные коды.

Процесс построения кодов Хаффмана состоит из нескольких шагов:

Шаг 1. Подсчет частоты встречаемости символов: Алгоритм анализирует данные и подсчитывает количество вхождений каждого символа.

Шаг 2. Построение дерева: Создается двоичное дерево, называемое деревом Хаффмана, где каждый узел представляет символ, а листья - это символы, которые необходимо закодировать. Вес каждого узла равен сумме весов его дочерних узлов.

Шаг 3. Кодирование: Идем по дереву от корня к каждому листу, присваивая "0" для левого ребенка и "1" для правого ребенка. Код символа определяется последовательностью "0" и "1" на пути от корня до листа.

Шаг 4. Создание таблицы: Создается таблица, в которой каждый символ связывается с его кодом Хаффмана. Эта таблица будет использоваться для кодирования данных.

Преимущество алгоритма Хаффмана заключается в том, что он генерирует оптимальные коды сжатия, которые максимально уменьшают размер данных. Он эффективен для сжатия текстовых файлов, а также других типов данных, где символы несколько предсказуемы и некоторые символы встречаются чаще других.

Проблема с остаточными битами решается с помощью специального символа выхода, который добавляется в конец закодированного сообщения. Этот символ служит для определения конца сообщения.

Во время кодирования, при окончании обработки исходного сообщения, добавляется кодировка для символа выхода. Во время декодирования, когда декодер достигает символа выхода, он прекращает декодирование и завершает работу.

Таким образом, остаточные биты, которые могут остаться в буфере кодировщика или декодера, не являются проблемой, поскольку символ выхода гарантированно присутствует в закодированном сообщении и используется для корректного завершения декодирования.

Также были реализованы буферы для чтения и записи битов в файлы.

Класс для чтения битов из входного потока данных принимает ссылку на объект типа std::istream в конструкторе и хранит внутри себя буфер и счетчик для отслеживания текущей позиции чтения. Метод Get используется для чтения определенного количества битов из потока. Если запрашиваемое количество битов превышает количество доступных в буфере, происходит чтение из потока и заполнение буфера. Метод Empty проверяет, пуст ли буфер и достигнут ли конец потока.

Класс для записи битов в выходной поток данных принимает ссылку на объект типа std::ostream в конструкторе и хранит внутри себя буфер и счетчик для отслеживания текущей позиции записи. Методы Add используются для добавления различных типов данных (uint8_t, uint16_t, uint32_t, std::vector<bool>, bool) в буфер.

Данные классы предоставляют удобный интерфейс для работы с битами.

Программа поддерживает различные ключи запуска:

1) -c : Если этот флаг установлен, то вывод результата сжатия будет отправлен на стандартный вывод вместо сохранения в файл.
2) -d : Если этот флаг установлен, то программа будет выполнять декомпрессию файлов вместо сжатия. Файлы должны иметь расширение ".z" для декомпрессии.
3) -k : Если этот флаг установлен, то при сжатии файлов будет сохранена копия исходного файла. Исходный файл будет иметь то же имя, но без расширения ".z".
4) -l : Если этот флаг установлен, программа выведет информацию о сжатом файле, включая его размер до и после сжатия, коэффициент сжатия и имя исходного файла. Файл должен иметь расширение ".z".
5) -r : Если этот флаг установлен, программа будет рекурсивно обрабатывать все файлы и подкаталоги в указанном каталоге. При декомпрессии файлы должны иметь расширение ".z".
6) -1: Если этот флаг установлен, программа будет использовать алгоритм сжатия LZW с максимальной длиной кода, равной 12. При превышении лимитной длины кода происходит сброс словаря. По умолчанию длина кода равна 14.
7) -9 : Если этот флаг установлен, программа будет использовать алгоритм сжатия LZW с максимальной длиной кода, равной 16. При превышении лимитной длины кода происходит сброс словаря. По умолчанию длина кода равна 14.
8) -t : Если этот флаг установлен, программа проверит, является ли указанный файл корректным сжатым файлом. Файл должен иметь расширение ".z". Ключом проверяется наличие символа выхода в

сжатом файле. Если файл является корректным сжатым файлом, будет выведено сообщение "The file is correct", в противном случае - "The file is incorrect".

9) - : Если этот флаг установлен, то ввод и вывод осуществляется в консоль. Данный флаг имеет приоритет над всеми другими.

При отсутствии ключей файл будет кодироваться, по умолчанию максимальная длина кода в алгоритме LZW равна 14, результат будет сохранён в файл, начальный файл будет удалён.

3. Текст программы

lzw.hpp

```
#include <iostream>
#include <vector>
#include <fstream>
#include <unordered_map>
#include <string>
#include <bitset>
#include "buffer.hpp"

struct TrieNode {
    std::unordered_map<uint8_t, TrieNode*> children;
    int index;

    TrieNode() {
        index = -1;
    }

    ~TrieNode() {
        for (auto& child : children) {
            delete child.second;
            child.second = nullptr;
        }
    }
};

class Trie {
    public:
        TrieNode* root;
```

```
int next_index;
uint32_t code_length;
bool flag;

Trie() {
    root = new TrieNode();
    next_index = 0;
    code_length = 9;
    flag = 0;
}

~Trie() {
    delete root;
}

void deleteTrie(TrieNode* node) {
    if (node == nullptr) {
        return;
    }

    for (auto& child : node->children) {
        deleteTrie(child.second);
        child.second = nullptr;
    }

    delete node;
    node = nullptr;
}

void Insert(const std::string& line) {
    TrieNode* node = root;

    for (int index = 0; index < line.length(); index++) {
        if (node->children.find((uint8_t)line[index]) == node->children.end()) {
            node->children[(uint8_t)line[index]] = new TrieNode();

            if (next_index == pow(2, code_length)) {
                code_length++;
            }
        }
```

```
            node = node->children[(uint8_t)line[index]];
        }

        node->index = next_index++;
    }

    int GetIndex(const std::string& line) {
        TrieNode* node = root;

        for (int index = 0; index < line.length(); index++) {
            if (node->children.find((uint8_t)line[index]) == node->children.end()) {
                return -1;
            }

            node = node->children[(uint8_t)line[index]];
        }

        return node->index;
    }

    std::string GetStringByIndex(int index) {
        return GetString(root, "", index);
    }

private:
    std::string GetString(TrieNode* node, const std::string& current, int index_number) {
        if (node->index == index_number) {
            return current;
        }

        std::string result;

        for (const auto& value : node->children) {
            result = GetString(value.second, current + std::string(1, static_cast<char>(value.first)), index_number);

            if (result != "") {
                return result;
            }
        }

        return "";
```

```
        }
};

void Compress(std::istream& input, std::ostream& output, uint32_t level) {
    Trie trie;
    std::string current, ending = "", concatenation;
    int index;
    OutputBuffer buffer(output);
    buffer.Add(level, 32);
    uint32_t stop = 0;
    uint8_t element, symbol = 0x0;
    input.peek();
    trie.Insert(ending);

    for (uint32_t counter = 0; counter <= 0xff; symbol++, counter++) {
        trie.Insert(std::string(1, static_cast<char>(symbol)));
    }

    while (!input.eof()) {
        input.read((char*)&element, 1);
        concatenation = current;
        concatenation += std::string(1, static_cast<char>(element));
        index = trie.GetIndex(concatenation);

        if (index != -1) {
            current = concatenation;
        } else {
            buffer.Add(trie.GetIndex(current), trie.code_length);
            trie.Insert(concatenation);
            current = std::string(1, static_cast<char>(element));

            if (trie.code_length > level) {
                trie.deleteTrie(trie.root);
                trie.root = new TrieNode();
                trie.next_index = 0;
                trie.code_length = 9;
                trie.Insert(ending);

                for (uint32_t index = 0; index <= 0xff; symbol++, index++) {
                    trie.Insert(std::string(1, static_cast<char>(symbol)));
                }
```

```
            }
        }

        input.peek();
    }

    if (!current.empty()) {
        buffer.Add(trie.GetIndex(current), trie.code_length);
    }

    buffer.Add(stop, trie.code_length);
    buffer.Cleaning();
}

int Decompress(std::istream& input, std::ostream& output) {
    Trie trie;
    std::string current = "", symbol, concatenation, ending = "", check;
    InputBuffer buffer(input);
    int level = buffer.Get(32), code, index;
    uint32_t stop = 0;

    if (!(level == 12 || level == 14 || level == 16)) {
        return 1;
    }

    trie.Insert(ending);

    for (uint32_t symbol = 0; symbol <= 0xff; symbol++) {
        trie.Insert(std::string(1, static_cast<char>(symbol)));
    }

    while (!buffer.Empty()) {
        if (trie.next_index == pow(2, trie.code_length)) {
            code = buffer.Get(trie.code_length + 1);
        } else {
            code = buffer.Get(trie.code_length);
        }

        if (code == stop) {
            break;
        }
```

```
            check = trie.GetStringByIndex(code);

            if (check != "") {
                symbol = check;
            } else {
                symbol = current + current[0];
            }

            concatenation = current + symbol[0];
            index = trie.GetIndex(concatenation);

            if (index != -1) {
                current = concatenation;
            } else {
                output.write(current.data(), current.length());
                trie.Insert(concatenation);
                current = symbol;

                if (trie.next_index + 1 > pow(2, level)) {
                    trie.deleteTrie(trie.root);
                    trie.root = new TrieNode();
                    trie.next_index = 0;
                    trie.code_length = 9;
                    trie.Insert(ending);
                    uint8_t insert_symbol = 0x0;

                    for (uint32_t counter = 0; counter <= 0xff; insert_symbol++, counter++) {
                        trie.Insert(std::string(1, static_cast<char>(insert_symbol)));
                    }

                    output.write(current.data(), current.length());
                    current = "";
                }
            }
        }

        output.write(current.data(), current.length());

        return 0;
    }
```

huffman.hpp

```
#include <fstream>
#include <iostream>
#include <vector>
#include <algorithm>
#include <queue>
#include <bitset>
#include <unordered_map>
#include "buffer.hpp"

class TreeNode {
    public:
        uint16_t symbol;
        uint32_t frequency;
        bool leaf;
        TreeNode* left;
        TreeNode* right;

        TreeNode() {
            symbol = 0;
            frequency = 0;
            leaf = 0;
            left = nullptr;
            right = nullptr;
        }

        TreeNode(uint16_t element) {
            symbol = element;
            frequency = 0;
            leaf = 1;
            left = nullptr;
            right = nullptr;
        }

        TreeNode(uint16_t element, uint32_t periodicity) {
            symbol = element;
            frequency = periodicity;
            leaf = 1;
            left = nullptr;
            right = nullptr;
```

```
        }

        TreeNode(TreeNode* left_node, TreeNode* right_node) {
            symbol = 0;
            frequency = left_node->frequency + right_node->frequency;
            leaf = 0;
            left = left_node;
            right = right_node;
        }

        virtual ~TreeNode() {
            if (!leaf) {
                delete left;
                delete right;
            }
        }
};

void DFS(const TreeNode* node, std::unordered_map<uint16_t, std::vector<bool>>& codes, std::vector<bool>&
code) {
    if (node->leaf) {
        codes[node->symbol] = code;
        return;
    }

    code.push_back(0);
    DFS(node->left, codes, code);
    code.pop_back();
    code.push_back(1);
    DFS(node->right, codes, code);
    code.pop_back();
}

void GetCodes(TreeNode* node, std::unordered_map<uint16_t, std::vector<bool>>& codes) {
    std::vector<bool> code;
    DFS(node, codes, code);
}

void DFSSerialize(const TreeNode* node, OutputBuffer& buffer) {
    if (node->leaf) {
        buffer.Add((bool)1);
```

```
        buffer.Add(node->symbol);
        return;
    }

    buffer.Add((bool)0);
    DFSSerialize(node->left, buffer);
    DFSSerialize(node->right, buffer);
}

void SerializeTree(TreeNode* root, OutputBuffer& buffer) {
    if (root == nullptr) {
        return;
    }

    DFSSerialize(root, buffer);
}

TreeNode* DFSDeserialize(InputBuffer& buffer) {
    if (!buffer.Get(1)) {
        TreeNode* node = new TreeNode();
        node->left = DFSDeserialize(buffer);
        node->right = DFSDeserialize(buffer);
        return node;
    } else {
        return new TreeNode((uint16_t)buffer.Get(16));
    }
}

TreeNode* DeserializeTree(InputBuffer& buffer) {
    return DFSDeserialize(buffer);
}

int Encode(std::istream& input, std::ostream& output) {
    std::vector<uint32_t> frequencies(0xff + 1, 0);
    uint8_t byte;
    uint16_t symbol = 0, exit_code = 0xff + 1;

    while (!input.eof()) {
        input.read((char*)&byte, 1);
        frequencies[byte]++;
        input.peek();
```

```
}

std::priority_queue<TreeNode*, std::vector<TreeNode*>, std::function<bool(TreeNode*, TreeNode*)>> queue(
    [](TreeNode* first_node, TreeNode* second_node) {
        return first_node->frequency > second_node->frequency;
    }
);

for (uint32_t index = 0; index < frequencies.size(); index++, symbol++) {
    if (frequencies[index]) {
        queue.push(new TreeNode(symbol, frequencies[index]));
    }
}

queue.push(new TreeNode(exit_code, 0));
frequencies.clear();
TreeNode* left_node, *rigth_node;

while (queue.size() > 1) {
    left_node = queue.top();
    queue.pop();
    rigth_node = queue.top();
    queue.pop();
    queue.push(new TreeNode(left_node, rigth_node));
}

TreeNode* root = queue.top();
queue.pop();
std::unordered_map<uint16_t, std::vector<bool>> code_table(0xff + 2);
GetCodes(root, code_table);
OutputBuffer buffer(output);
SerializeTree(root, buffer);
input.clear();
input.seekg(0);

while (!input.eof()) {
    input.read((char*)&byte, 1);
    buffer.Add(code_table[byte]);
    input.peek();
}
```

```
    buffer.Add(code_table[exit_code]);
    buffer.Cleaning();
    code_table.clear();
}

void Decode(std::istream& input, std::ostream& output) {
    InputBuffer buffer(input);
    TreeNode* root = DeserializeTree(buffer);
    TreeNode* node = root;
    uint16_t exit_code = 0xff + 1;

    while (!buffer.Empty()) {
        if (buffer.Get(1)) {
            node = node->right;
        } else {
            node = node->left;
        }

        if (node->leaf) {
            if (node->symbol == exit_code) {
                return;
            }

            output.write((char*)&node->symbol, 1);
            node = root;
        }
    }
}
```

buffer.hpp

```
#ifndef BITSBUFFER_HPP
#define BITSBUFFER_HPP

#include <fstream>
#include <iostream>
#include <vector>
#include <algorithm>

class InputBuffer {
    public:
        InputBuffer(std::istream& input): file(input) {
```

```
    buffer = 0;
    counter = 0;
}

uint32_t Get(uint32_t length) {
    uint32_t symbol;

    if (length > counter) {
        Read(length);
    }

    if (length > counter) {
        uint32_t storage = buffer;
        file.peek();
        buffer = 0;
        counter = 0;
        return storage;
    }

    counter -= length;
    symbol = (uint32_t)(buffer >> counter);

    if (counter != 0) {
        buffer = buffer << (64 - counter);
        buffer = buffer >> (64 - counter);
    } else {
        symbol = buffer;
        buffer = 0;
    }

    return symbol;
}

bool Empty() {
    if (counter == 0 && file.peek() == EOF) {
        return true;
    }
    return false;
}

void Cleaning() {
```

```
            buffer = 0;
            counter = 0;
        }

    private:
        std::istream& file;
        uint64_t buffer;
        uint32_t counter;

        void Read(uint32_t length) {
            uint8_t byte;

            while (length > counter) {
                file.read((char*)&byte, 1);
                counter += 8;
                buffer = (buffer << 8) | (uint64_t)(byte);
                file.peek();
            }
        }
};

class OutputBuffer {
    public:
        OutputBuffer(std::ostream& output) : file(output) {
            buffer = 0;
            counter = 0;
            }

        void Add(uint8_t code) {
            counter += 8;
            buffer = (buffer << 8) | (uint64_t)code;
            Write();
        }

        void Add(uint16_t code) {
            counter += 16;
            buffer = (buffer << 16) | (uint64_t)code;
            Write();
        }

        void Add(uint32_t code, uint32_t length) {
```

```
    counter += length;
    buffer = (buffer << length) | (uint64_t)code;

    if (counter >= 24) {
        Write();
    }
}

void Add(std::vector<bool>& code) {
    for (int index = 0; index < code.size(); index++) {
        bool value = code[index];

        if (!value) {
            buffer = (buffer << 1);
        } else {
            buffer = (buffer << 1) | 1;
        }

        if (counter++ >= 32) {
            Write();
        }
    }
}

void Add(bool code) {
    if (!code) {
        buffer = (buffer << 1);
    } else {
        buffer = (buffer << 1) | 1;
    }

    if (counter++ >= 24) {
        Write();
    }
}

void Cleaning() {
    Write();
    uint8_t byte = 0;

    if (counter == 0) {
```

```
            return;
        }

        byte = (uint8_t)(buffer << (8 - counter));
        file.write((char*)&byte, 1);
        buffer = 0;
        counter = 0;
    }

    virtual ~OutputBuffer() {}

private:
    std::ostream& file;
    uint64_t buffer;
    uint32_t counter;

    void Write() {
        uint8_t byte;

        while (counter >= 8) {
            counter -= 8;
            byte = (uint8_t)(buffer >> (counter));

            if (counter != 0) {
                buffer = buffer << (64 - counter);
                buffer = buffer >> (64 - counter);
            } else {
                buffer = 0;
            }

            file.write((char*)&byte, 1);
        }
    }
};

#endif
```

archiver.cpp

```
#include <cstdio>
#include <getopt.h>
#include <dirent.h>
```

```
#include <sys/stat.h>
#include <fstream>
#include <cmath>
#include <sstream>
#include <experimental/filesystem>
#include "lzw.hpp"
#include "huffman.hpp"

int FileProccessing(std::string filename, uint32_t level, bool decompress, bool output_preservation, bool preservation,
bool stdcin) {
    std::string result, name = filename + ".tmp";
    std::ifstream input, input_file;
    std::ofstream output, output_file;
    uint32_t pointer = 0;
    double original_size, final_size, coefficient;

    if (decompress && !stdcin) {
        output_file.open(name, std::ofstream::out | std::ofstream::binary);

        if (!(output_file.is_open())) {
            std::cerr << "Error while opening file " << name << '\n';
            output_file.close();
            return 1;
        }

        if (filename.substr(filename.length() - 2) != ".z") {
            std::cerr << "The file name must end with .z" << '\n';
            return 1;
        }

        input.open(filename, std::ifstream::in | std::ifstream::binary);

        if (!(input.is_open())) {
            std::cerr << "Error while opening file " << filename << std::endl;
            input.close();
            return 1;
        }

        uint8_t byte = 0xee;
        uint8_t byte_read;
        input.clear();
```

```
    input.read((char *)&byte_read, 1);

    if (byte_read == byte) {
        Decode(input, output_file);
    } else {
        pointer = 1;
    }

    input.close();
    output_file.close();

    if (pointer) {
        return pointer;
    }

    input_file.open(name, std::ifstream::in | std::ifstream::binary);

    if (!(input_file.is_open())) {
        std::cerr << "Error while opening file " << name << std::endl;
        input_file.close();
        return 1;
    }

    if (output_preservation) {
        std::ostream &standart_output = std::cout;
        Decompress(input_file, standart_output);
    } else {
        output.open(filename.substr(0, filename.length() - 2), std::ofstream::out | std::ofstream::binary);

        if (!(output.is_open())) {
            std::cerr << "Error while opening file " << filename.substr(0, filename.length() - 2) << std::endl;
            output.close();
            return 1;
        }

        pointer = Decompress(input_file, output);
        output.close();
    }

    input_file.close();
} else {
```

```
output_file.open(name, std::ofstream::out | std::ofstream::binary);

if (!(output_file.is_open())) {
        std::cerr << "Error while opening file " << name << '\n';
        output_file.close();
        return 1;
    }

if (!stdcin) {
    input.open(filename, std::ifstream::in | std::ifstream::binary);

    if (!(input.is_open())) {
        std::cerr << "Error while opening file " << filename << std::endl;
        input.close();
        return 1;
    }

    original_size = input.tellg();
    Compress(input, output_file, level);
    input.close();
} else {
    std::istream &standart_input = std::cin;
    Compress(standart_input, output_file, level);
}

output_file.close();
input_file.open(name, std::ifstream::in | std::ifstream::binary);

if (!(input_file.is_open())) {
    std::cerr << "Error while opening file " << name << std::endl;
    input_file.close();
    return 1;
}

if (output_preservation || stdcin) {
    std::ostream &standart_output = std::cout;
    uint8_t byte = 0xee;
    standart_output.write((char *)&byte, 1);
    Encode(input_file, standart_output);
} else {
    output.open(filename + ".z", std::ofstream::out | std::ofstream::binary);
```

```
            if (!(output.is_open())) {
                std::cerr << "Error while opening file " << filename + ".z" << std::endl;
                output.close();
                return 1;
            }

            uint8_t byte = 0xee;
            output.write((char *)&byte, 1);
            Encode(input_file, output);
            output.write((char *)&byte, 1);
            uint64_t size = std::experimental::filesystem::file_size(filename);
            output.write((char *)&size, 8);
            output.close();
        }

        input_file.close();
    }

    if (!output_preservation && !preservation && !stdcin) {
        std::remove(filename.c_str());
    }

    if (decompress) {
        return pointer;
    } else {
        return 0;
    }
}

void ProcessDirectory(const std::string& directory, uint32_t level, bool decompress, bool recursive, bool
output_preservation, bool preservation, bool stdcin) {
    DIR* storage;
    struct dirent* entry;

    if ((storage = opendir(directory.c_str())) != nullptr) {
        while ((entry = readdir(storage)) != nullptr) {
            std::string filename = entry->d_name;
            std::string full_path = directory + "/" + filename;

            if (filename == "." || filename == ".." || filename.rfind(".", 0) == 0) {
```

```
            continue;
        }

        struct stat structure;
        if (stat(full_path.c_str(), &structure) != -1 && S_ISDIR(structure.st_mode)) {
            if (recursive) {
            ProcessDirectory(full_path, level, decompress, recursive, output_preservation, preservation, stdcin);
            }
        } else {
            if (decompress) {
                FileProccessing(full_path, level, decompress, output_preservation, preservation, stdcin);
            } else {
                FileProccessing(full_path, level, decompress, output_preservation, preservation, stdcin);
            }
        }
    }

    closedir(storage);
  } else {
    std::cerr << "Error opening directory: " << directory << '\n';
  }
}

int main(int argc, char* argv[]) {
  const char* short_options = "cdklr19t";
  const struct option long_options[] = {
    {"output_preservation", no_argument, nullptr, 'c'},
    {"decompress", no_argument, nullptr, 'd'},
    {"preservation", no_argument, nullptr, 'k'},
    {"information", no_argument, nullptr, 'l'},
    {"recursive", no_argument, nullptr, 'r'},
    {"level1", no_argument, nullptr, '1'},
    {"level9", no_argument, nullptr, '9'},
    {"test", no_argument, nullptr, 't'},
    {nullptr, 0, nullptr, 0}
  };

  bool output_preservation = false;
  bool decompress = false;
  bool preservation = false;
  bool information = false;
```

```
bool recursive = false;
bool level1 = false;
bool level9 = false;
bool test = false;
bool stdcin = false;
std::string filename;
int option;

while ((option = getopt_long(argc, argv, short_options, long_options, nullptr)) != -1) {
    switch (option) {
        case 'c':
            output_preservation = true;
            break;
        case 'd':
            decompress = true;
            break;
        case 'k':
            preservation = true;
            break;
        case 'l':
            information = true;
            break;
        case 'r':
            recursive = true;
            break;
        case '1':
            level1 = true;
            break;
        case '9':
            level9 = true;
            break;
        case 't':
            test = true;
            break;
        default:
            std::cerr << "Unknown option: " << optarg << '\n';
            return 1;
    }
}

if (optind < argc) {
```

```
        filename = argv[optind];

        if (filename == "-") {
            stdcin = true;
        }
    } else {
        std::cerr << "No filename provided" << '\n';
        return 1;
    }

    if (information && !output_preservation && !decompress && !preservation && !recursive && !level1 && !level9 &&
!test && !stdcin) {
        if (filename.substr(filename.length() - 2) == ".z") {
            std::ifstream file;
            double coefficient;
            uint64_t original_size, final_size = std::experimental::filesystem::file_size(filename);
            file.open(filename, std::ifstream::in | std::ifstream::binary);

            if (!(file.is_open())) {
                std::cerr << "Error while opening file " << filename << std::endl;
                file.close();
                return 1;
            }

            file.seekg(-8, file.end);
            file.read((char *)&original_size, 8);
            file.close();
            std::cout << "Options:\n";
            std::cout << "Uncompressed file size: " << original_size << '\n';
            std::cout << "Compressed file size: " << final_size << '\n';

            if (original_size > final_size) {
                coefficient = (round((((double)original_size - (double)final_size) / (double)original_size) * 1000.0)) / 10.0;
            } else {
                coefficient = 0;
            }

            std::cout << "Ratio: " << coefficient << "%\n";
            std::cout << "Uncompressed file name: " << filename.substr(0, filename.length() - 2) << '\n';
            return 0;
        } else {
```

```
            std::cerr << "The file name must end with .z" << '\n';
            return 1;
        }
    }

    if (test && !output_preservation && !decompress && !preservation && !recursive && !level1 && !level9 &&
!information && !stdcin) {
        if (filename.substr(filename.length() - 2) == ".z") {
            std::ifstream file;
            uint8_t byte;
            bool flag = false;
            file.open(filename, std::ifstream::in | std::ifstream::binary);

            if (!(file.is_open())) {
                std::cerr << "Error while opening file " << filename << std::endl;
                file.close();
                return 1;
            }

            file.read((char *)&byte, 1);

            if (byte != 0xee) {
                flag = true;
            } else {
                file.seekg(-9, file.end);
                file.read((char *)&byte, 1);

                if (byte != 0xee) {
                    flag = true;
                }
            }

            file.close();

            if (flag) {
                std::cout << "The file is incorrect" << '\n';
            } else {
                std::cout << "The file is correct" << '\n';
            }

            return 0;
```

```
        } else {
            std::cerr << "The file name must end with .z" << '\n';
            return 1;
        }
    }

    uint32_t level = 14;

    if (level1) {
        level = 12;
    } else if (level9) {
        level = 16;
    }

    if (recursive) {
        ProcessDirectory(filename, level, decompress, recursive, output_preservation, preservation, stdcin);
    } else {
        FileProccessing(filename, level, decompress, output_preservation, preservation, stdcin);
    }

    return 0;
}
```

4. Демонстрация работы программы

MacBook-Pro-MacBook:~ macbookpro$ cat >input.txt
abracadabra
^C
MacBook-Pro-MacBook:~ macbookpro$ ./archiver input.txt
MacBook-Pro-MacBook:~ macbookpro$ ./archiver -d input.txt.z
MacBook-Pro-MacBook:~ macbookpro$ cat input.txt
abracadabra
MacBook-Pro-MacBook:~ macbookpro$

Тестирование на тексте произведения «Слово о полку Игореве»:
MacBook-Pro-MacBook:~ macbookpro$ ./archiver -9 input7.txt
MacBook-Pro-MacBook:~ macbookpro$ ./archiver -k -d input7.txt.z
MacBook-Pro-MacBook:~ macbookpro$ ./archiver -t input7.txt.z
The file is correct
MacBook-Pro-MacBook:~ macbookpro$ ./archiver -l input7.txt.z
Options:
Uncompressed file size: 39416

Compressed file size: 15377
Ratio: 61%
Uncompressed file name: input7.txt
MacBook-Pro-MacBook:~ macbookpro$

5. Тест производительности

Анализ степени сжатия:

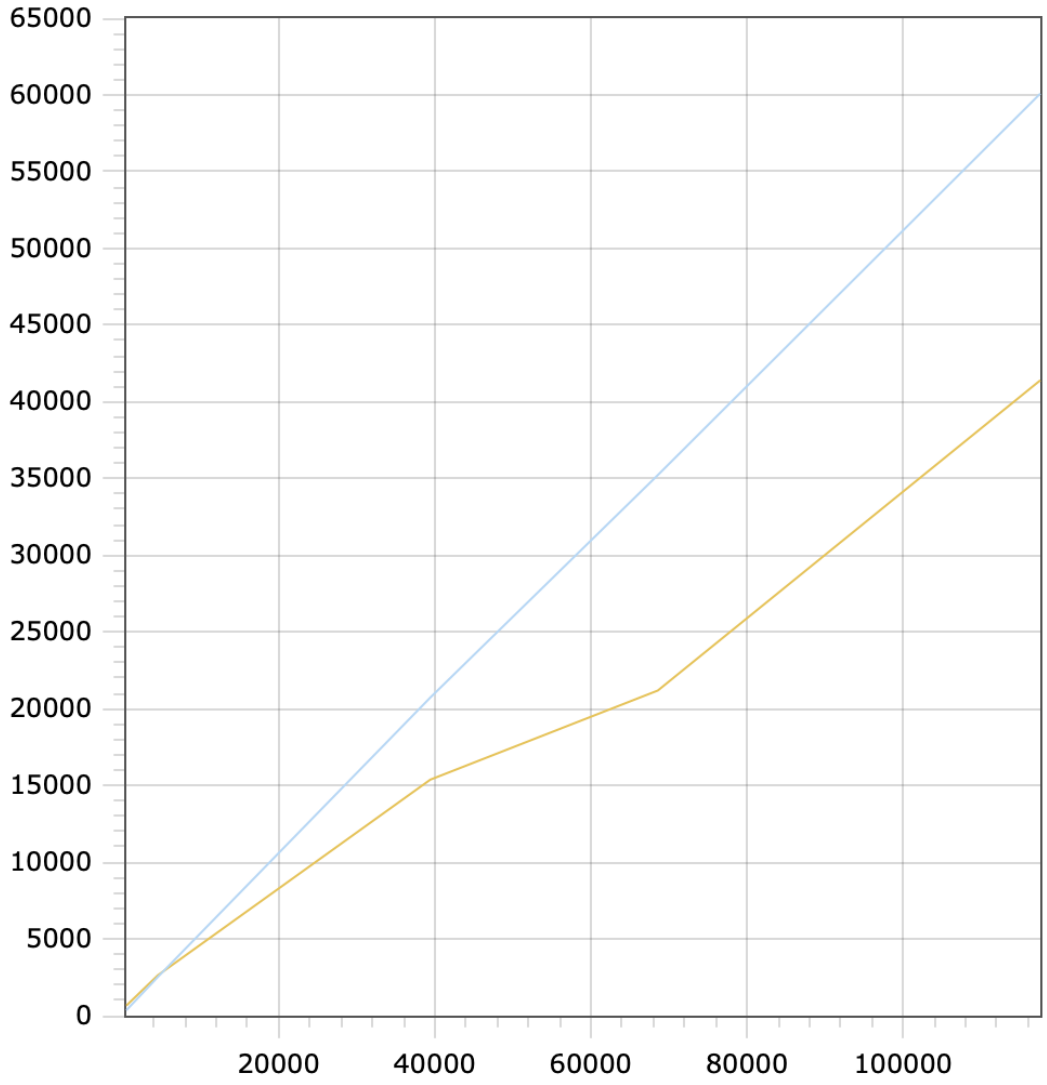
Тестирование на текстовых файлах (без использования ключей):

LZW + Huffman:

Название произведения	Размер начального файла (байт)	Размер конечного файла (байт)	Сжатие (%)
С.А.Есенин «Береза»	473	653	0
М.Ю.Лермонтов «Бородино»	4499	2627	41,6
«Слово о полку Игореве»	39416	15377	61
И.С.Тургенев «Бежин луг»	68575	21188	64,7
М.Ю.Лермонтов «Герой нашего времени» первая часть	117572	41401	64,8

Huffman:

Название произведения	Размер начального файла (байт)	Размер конечного файла (байт)	Сжатие (%)
С.А.Есенин «Береза»	473	350	26
М.Ю.Лермонтов «Бородино»	4499	2515	44,1
«Слово о полку Игореве»	39416	20728	47,4
И.С.Тургенев «Бежин луг»	68575	35229	48,6
М.Ю.Лермонтов «Герой нашего времени» первая часть	117572	60085	48,9



Оранжевый график – LZW + Huffman, голубой график – Huffman.

Тестирование на изображениях (без использования ключей):

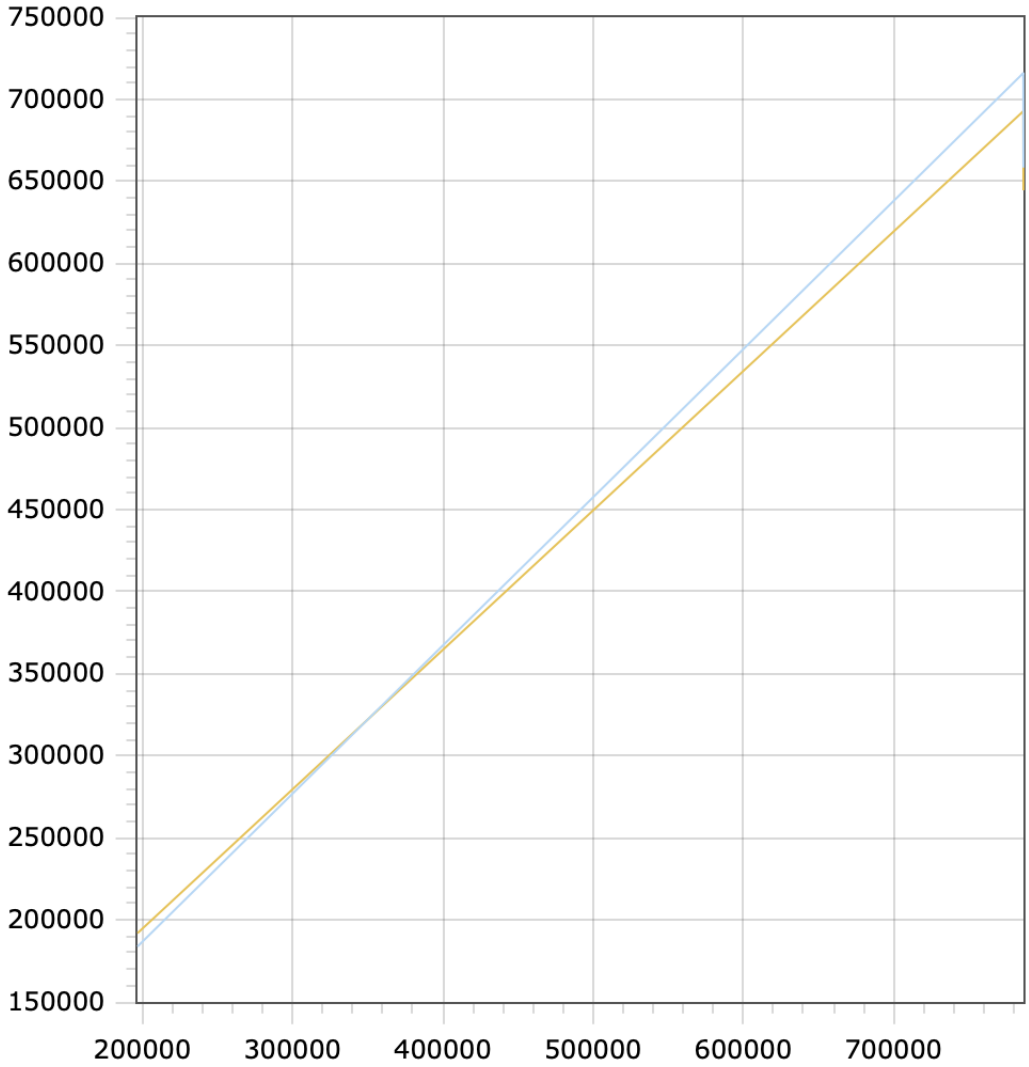
LZW + Huffman:

Размер начального файла (байт)	Размер конечного файла (байт)	Сжатие (%)
196662	191903	2,4
786400	692757	11,9
786486	644575	18

Huffman:

Размер начального файла (байт)	Размер конечного файла (байт)	Сжатие (%)
196662	183753	6,6

786400	715979	9
786486	658393	16,3



Оранжевый график – LZW + Huffman, голубой график – Huffman.

По анализу степени сжатия файлов можно увидеть, что на файлах с маленьким размером эффективнее сжимает алгоритм Хаффмана, а файлы с большим размером лучше сжимает комбинация алгоритмов LZW и Хаффмана.

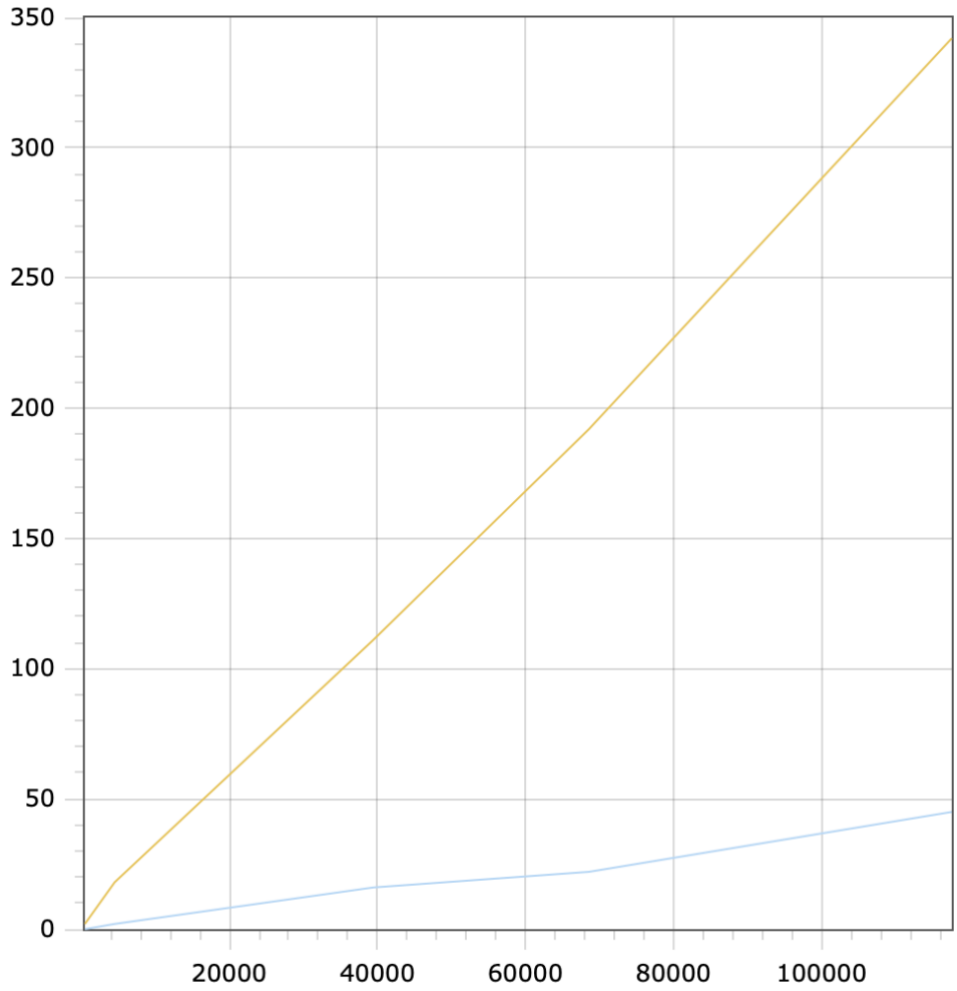
Анализ времени работы:

Тестирование на текстовых файлах (без использования ключей):

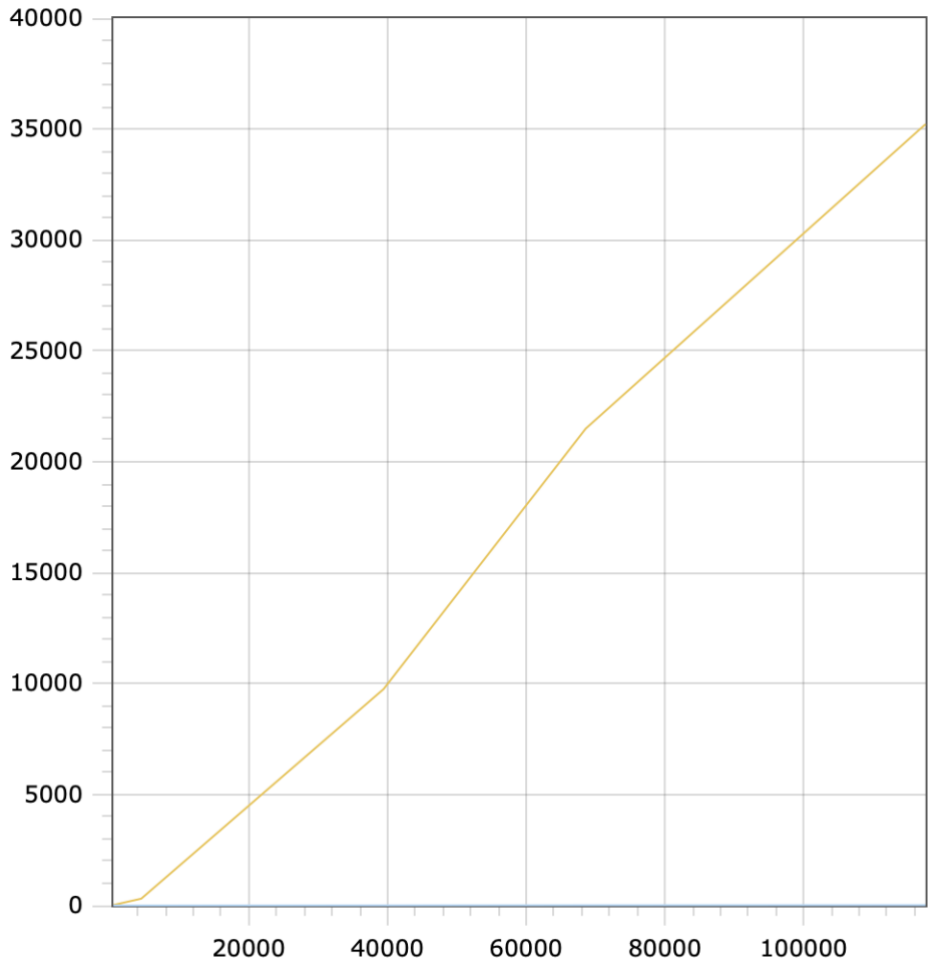
Размер начального файла (байт)	Время работы алгоритма LZW + Huffman (мс)	Время работы алгоритма Huffman (мс)
473	Кодирование: 2	Кодирование: 0

	Декодирование: 21	Декодирование: 0
4499	Кодирование: 18 Декодирование: 306	Кодирование: 2 Декодирование: 0
39416	Кодирование: 111 Декодирование: 9750	Кодирование: 16 Декодирование: 5
68575	Кодирование: 192 Декодирование: 21496	Кодирование: 22 Декодирование: 11
117572	Кодирование: 342 Декодирование: 35223	Кодирование: 45 Декодирование: 16

Кодирование:



Декодирование:

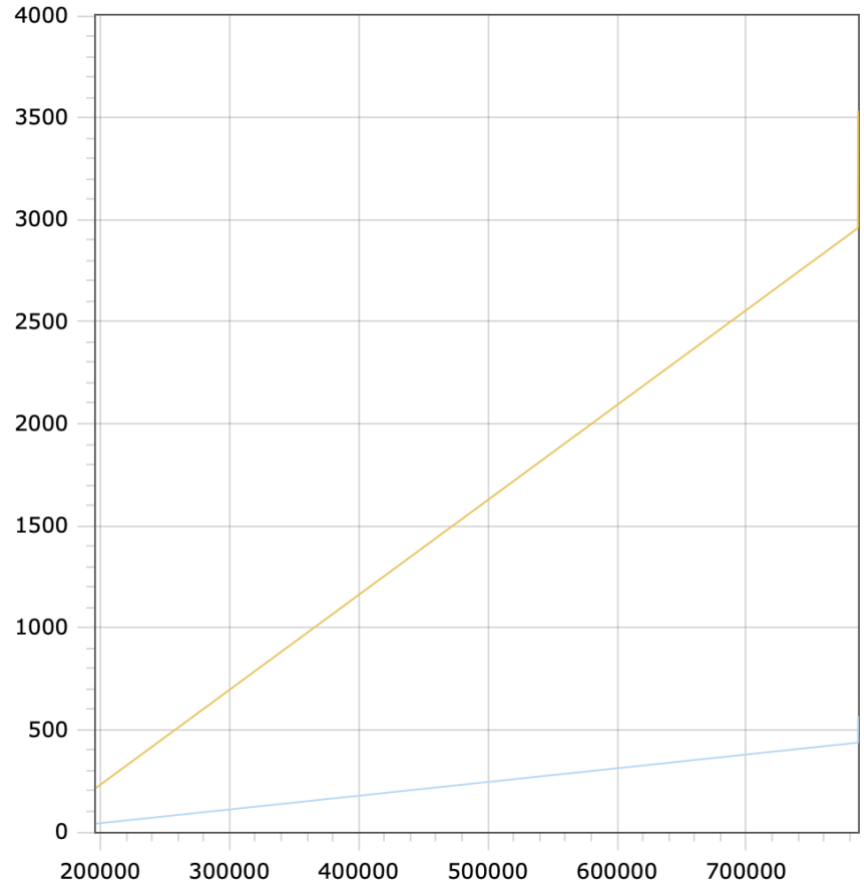


Оранжевый график – LZW + Huffman, голубой график – Huffman.

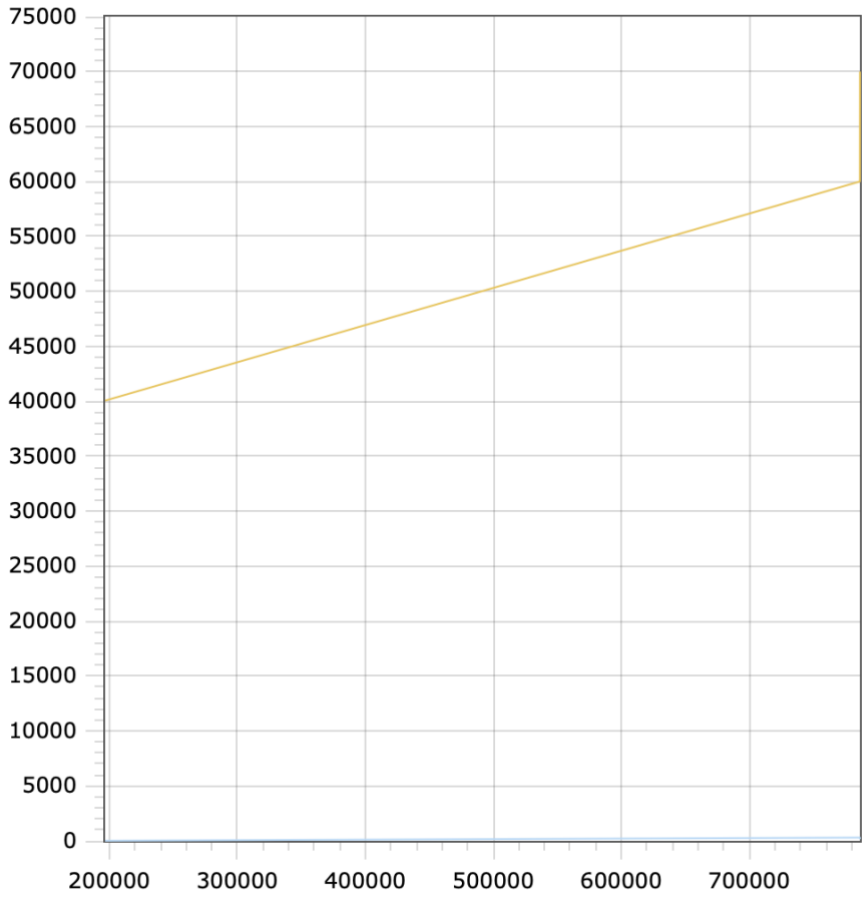
Тестирование на изображениях (без использования ключей):

Размер начального файла (байт)	Время работы алгоритма LZW + Huffman (мс)	Время работы алгоритма Huffman (мс)
196662	Кодирование: 215 Декодирование: 40058	Кодирование: 40 Декодирование: 11
786400	Кодирование: 2959 Декодирование: ∝	Кодирование: 437 Декодирование: 297
786486	Кодирование: 3535 Декодирование: ∝	Кодирование: 567 Декодирование: 164

Кодирование:



Декодирование:



Оранжевый график – LZW + Huffman, голубой график – Huffman.

По анализу времени работы алгоритмов можно увидеть, что применять просто алгоритм Хаффмана намного быстрее, чем его комбинацию с LZW. Выбор между алгоритмом и комбинацией алгоритмов нужно основывать на том, что в данном случае пользователю важнее – скорость кодирования и декодирования или объём памяти, занимаемый сжатым файлом.

6. Вывод

В результате выполнения лабораторной работы я познакомилась с алгоритмами сжатия Хаффмана и LZW, с помощью них реализовала архиватор, кодирующий и декодирующий файлы разных форматов, поддерживающий различные ключи, проанализировала время работы программы и степень сжатия.

7. Список литературы

1) Гасфилд Дэн «Строки, деревья, последовательности в алгоритмах: Информатика и вычислительная биология» - СПб.: Невский Диалект; БХВ-Петербург, 2003. – 654 с.
2) Кормен, Томас Х., Лейзерсон, Чарльз И., Ривест, Рональд Л., Штайн, Клиффорд «Алгоритмы: построение и анализ, 2-е издание. : Пер. с англ. – М. : Издательский дом «Вильямс», 2011. – 1296 с.
3) Алгоритм Лемпеля – Зива – Велча.

   URL: https://ru.wikipedia.org/wiki/Алгоритм_Лемпеля_—_Зива_—_Велча#Декодирование

   (дата последнего обновления 29.05.2023)
4) Алгоритм LZW.

   URL: https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_LZW

   (дата последнего обновления 04.09.2022)
5) Алгоритм Хаффмана на пальцах.

   URL: https://habr.com/ru/articles/144200

   (дата последнего обновления: 25.05.2012)