МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное

учреждение высшего образования

"МИРЭА - Российский технологический университет"

РТУ МИРЭА

Институт информационных технологий (ИТ)

Кафедра математического обеспечения и стандартизации информационных технологий (МОСИТ)

ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ № 8.1

по дисциплине

«Структуры и алгоритмы обработки данных»

Тема: Алгоритмы кодирования и сжатия данных

Выполнил студент группы ИКБО-41-23 Гольд Д.В.

Принял старший преподаватель Рысин М.Л.

Москва 2024

СОДЕРЖАНИЕ

ЗАДАНИЕ 1

ЦЕЛЬ:

Исследовать такие алгоритмы сжатия, как Шеннон-Фано, LZ77, LZ78

1. Закодировать фразу методами Шеннона–Фано

Эни-бени рити-Фати. Дорба, дорба сентибрати. Дэл. Дэл. Кошка. Дэл. Фати!

(72 символа)

Незакодированная фраза = 576 бит

Закодированная фраза = 285 бит

Символ	Кол-во	Вероятность	Код	Кол-во бит
	9	0.125	001	27
и	8	0.111	000	24
а	6	0.083	010	18
.	6	0.083	0111	24
т	5	0.0694	0110	20
б	4	0.055	1100	16
р	4	0.055	1101	16
Д	4	0.055	1110	16
н	3	0.0416	11110	15
о	3	0.0416	11111	15
л	3	0.0416	10000	12
-	2	0.027	10010	10
е	2	0.027	10011	10
Ф	2	0.027	101011	12

Э	1	0.0138	101010	6
д	1	0.0138	101000	6
,	1	0.0138	101001	6
ш	1	0.0138	101110	6
!	1	0.0138	101100	6
к	1	0.0138	101101	6
с	1	0.0138	1011110	7
К	1	0.0138	1011111	7

2. Сжатие данных по методу Лемпеля–Зива LZ77, используя двухсимвольный алфавит (0, 1) закодировать следующую фразу:

Исходный текст	0.00.10.01.011.001.000.100.01
LZ-код	0.00.010.001.1001.0101.0100.0110.0100
R	2 3 4
Вводимые коды	- 00 10 01 011 001 000 100

Таблица соответствия кодов и символов:

Символ	Код, R = 1	Код, R = 2	Код, R = 3	Код, R = 4
0	0	00	000	0000
1	1	01	001	0001
00		10	010	0010
10		11	011	0011
01			100	0100
011			101	0101
001			110	0110
000			111	0111
100				1000

3. Закодировать следующую фразу, используя код LZ78

kloklonkolonklonkl

Словарь	Считываемое содержимое	Код
	k	<0, k>
k = 1	l	<0, l>
k = 1, l = 2,	o	<0, o>
k = 1, l = 2, o = 3,	kl	<1, l>
k = 1, l = 2, o = 3, kl = 4	klo	<4, o>
k = 1, l = 2, o = 3, kl = 4, klo = 5	n	<0, n>
k = 1, l = 2, o = 3, kl = 4, klo = 5, n = 6,	ko	<1, o>
k = 1, l = 2, o = 3, kl = 4, klo = 5, n = 6, ko = 7	lo	<2, o>
k = 1, l = 2, o = 3, kl = 4, klo = 5, n = 6, ko = 7, lo = 8, nk = 9	nk	<6, k>
k = 1, l = 2, o = 3, kl = 4, klo = 5, n = 6, ko = 7, lo = 8, nk = 9	lon	<8, n>
k = 1, l = 2, o = 3, kl = 4, klo = 5, n = 6, ko = 7, lo = 8, nk = 9, lon = 10	kl	<4, EOF>

<0, k><0, l><0, o><1, l><4, o><0, n><1, o><2, o><6, k><8, n><4, EOF>

ЗАДАНИЕ 2

ЦЕЛЬ:

Разработать программы сжатия и восстановления текста методами Хаффмана и Шеннона–Фано.

Алгоритм Шеннона-Фано:

Формирование кодов:

- Символы сортируются по частоте вхождения в строку.
- Разделяются на группы с равной суммой вероятностей.
- Каждой группе присваиваются биты 0 и 1.
- Процесс повторяется до назначения кода каждому символу.

Код программы:

```cpp
#include <iostream>
#include <vector>
#include <map>
#include <string>
#include <algorithm>
#include <locale>
#include <codecvt>

using namespace std;

struct Node {//узел дерева шеннона-фано
    char symbol;
    double symbol_frequency;
    string character_code;
};

bool compare_frequency(Node tree_nodes_a, Node tree_nodes_b) {//сравниваем частот для сортировоки
    return tree_nodes_a.symbol_frequency > tree_nodes_b.symbol_frequency;
}

void generate_codes(vector<Node>& symbol_nodes, int start, int end, string character_code) {//рекурс функ - построение кодов
    if (start == end) {
        symbol_nodes[start].character_code = character_code;
        return;
    }

    double sum_symbol_frequency_left = 0, sum_symbol_frequency_right = 0;
    int left_froup_end_index  = start;

    for (int i = start; i <= end; ++i) sum_symbol_frequency_right += symbol_nodes[i].symbol_frequency;

    for (int i = start; i <= end; ++i) {
        sum_symbol_frequency_left += symbol_nodes[i].symbol_frequency;
        sum_symbol_frequency_right -= symbol_nodes[i].symbol_frequency;

        if (sum_symbol_frequency_left >= sum_symbol_frequency_right) {
            left_froup_end_index  = i;
            break;
        }
    }

    generate_codes(symbol_nodes, start, left_froup_end_index , character_code + "0");
    generate_codes(symbol_nodes, left_froup_end_index  + 1, end, character_code + "1");
}

int main() {
    setlocale(LC_ALL, "");

    //string input = "Эни-бени_рити-Фати._Дорба,_дорба_сентибрати._Дэл._Дэл._Кошка._Дэл._Фати!";
    //э = q ; Э = Q ; ш = s;

    string input = "Qni-beni riti-Fati. Dorba, dorba centibrati. Dql. Dql. Koska. Dql. Fati!";

    map<char, double> frequency_map;//частота симв
    for (char c : input) {
        frequency_map[c]++;
    }

    vector<Node> symbol_nodes;
    for (auto pair : frequency_map) {
        symbol_nodes.push_back({ pair.first, pair.second / input.size(), "" });
    }

    sort(symbol_nodes.begin(), symbol_nodes.end(), compare_frequency);

    generate_codes(symbol_nodes, 0, symbol_nodes.size() - 1, "");//строим дерево

    map<char, string> codes;
    for (auto node : symbol_nodes) {
```

```cpp
    map<char, string> codes;
    for (auto node : symbol_nodes) {
        codes[node.symbol] = node.character_code;
        cout << node.symbol << " -> " << node.character_code << endl;
    }

    string encoded = "";//кодирование
    for (char c : input) {
        encoded += codes[c];
    }

    cout << "encoded text: " << encoded << endl;

    //сжатый текст
    int original_size_bits = input.size() * 8;//исходный размер в битах
    int compressed_size_bits = encoded.size();//сжатый размер в битах

    double compression_ratio = (double)compressed_size_bits / original_size_bits * 100;//процент сжатия

    string decoded = "";//декодирование
    string temp = "";
    for (char bit : encoded) {
        temp += bit;
        for (auto node : symbol_nodes) {
            if (node.character_code == temp) {
                decoded += node.symbol;
                temp = "";
                break;
            }
        }
    }

    double average_code_length = 0.0;//вычисление средней длины кода
    for (auto node : symbol_nodes) {
```

```cpp
    double average_code_length = 0.0;//вычисление средней длины кода
    for (auto node : symbol_nodes) {
        average_code_length += node.character_code.length() * node.symbol_frequency;
    }

    double variance = 0.0;//вычисление дисперсии длины кода
    for (auto node : symbol_nodes) {
        double length_diff = node.character_code.length() - average_code_length;
        variance += pow(length_diff, 2) * node.symbol_frequency;
    }

    cout << "\ncompressed text in bits: " << compressed_size_bits << endl;
    cout << "compression ratio: " << compression_ratio << "%\n";
    cout << "\ndecoded text: " << decoded << endl;
    cout << "\naverage code length: " << average_code_length << endl;
    cout << "code length variance: " << variance << endl;

    return 0;
}
```

Тестирование:

```
  -> 0000
i -> 0001
. -> 001
a -> 0100
t -> 0101
D -> 011
b -> 10000
r -> 10001
l -> 1001
n -> 10100
o -> 10101
q -> 1011
- -> 11000
F -> 11001
e -> 11010
! -> 110110
, -> 110111
K -> 111000
Q -> 111001
c -> 111010
d -> 111011
k -> 11110
s -> 11111
encoded text: 11100110100000111000100001101010100000100001000100010101000111000110010100010100010010000011101011000110000100110111000011101110101100011000001000000111010110101010001010001100001000101000101000100100000111011100100100000111011100100100000111000101011111111111001000010000011101110010010000110010100010100011101100

compressed text in bits: 312
compression ratio: 54.1667%

decoded text: Qni-beni riti-Fati. Dorba, dorba centibrati. Dql. Dql. Koska. Dql. Fati!

average code length: 4.33333
code length variance: 0.666667
```

Алгоритм Хаффмана:

Составление списка узлов:

- Символы сортируются по частоте вхождения в строку.

Построение дерева:

- Найти два узла с наименьшими весами.
- Объединить их в новый узел с весом, равным сумме их частот.
- Новый узел становится родительским для этих двух.
- Повторять процесс, пока не останется один узел (корень дерева).

Назначение кодов:

- Пройти по дереву от корня к листьям.
- Каждой левой ветви присвоить 0, а правой — 1.
- Листовые узлы содержат символы с их уникальными бинарными кодами.

Код программы:

```cpp
#include <iostream>
#include <string>
#include <map>
#include <queue>
#include <vector>
#include <algorithm>

using namespace std;

struct huffman_node {//узел дерева хаффмана
    char symbol;
    double symbol_frequency;
    huffman_node* left_child;
    huffman_node* right_child;
};

struct compare {//компаратор для приоритетной очереди
    bool operator()(huffman_node* tree_nodes_a, huffman_node* tree_nodes_b) {
        return tree_nodes_a->symbol_frequency > tree_nodes_b->symbol_frequency;
    }
};

void generate_huffman_codes(huffman_node* node, string current_code, map<char, string>& huffman_codes) {
    if (!node) return;
    //рекурсивная функция для генерации кодов символов

    if (!node->left_child && !node->right_child) {//если узел листовой - добавляем символ и его код
        huffman_codes[node->symbol] = current_code;
    }

    //рекурсивно вызываем для левого и правого поддерева
    generate_huffman_codes(node->left_child, current_code + "0", huffman_codes);
    generate_huffman_codes(node->right_child, current_code + "1", huffman_codes);
}

void delete_huffman_tree(huffman_node* node) {//рекурсивное удаление дерева
    if (!node) return;
    delete_huffman_tree(node->left_child);
    delete_huffman_tree(node->right_child);
```

```cpp
void delete_huffman_tree(huffman_node* node) {//рекурсивное удаление дерева
    if (!node) return;
    delete_huffman_tree(node->left_child);
    delete_huffman_tree(node->right_child);
    delete node;
}

int main() {
    string input_text = "Gold Daniil Vladimirovich";

    map<char, double> frequency_map;
    for (char c : input_text) {
        frequency_map[c]++;
    }

    //создание приоритетной очереди
    priority_queue<huffman_node*, vector<huffman_node*>, compare> priority_queue_huffman_nodes;
    for (auto pair : frequency_map) {
        huffman_node* new_huffman_node = new huffman_node{ pair.first, pair.second, nullptr, nullptr };
        priority_queue_huffman_nodes.push(new_huffman_node);
    }

    //построение дерева хаффмана
    while (priority_queue_huffman_nodes.size() > 1) {
        huffman_node* left_child = priority_queue_huffman_nodes.top(); priority_queue_huffman_nodes.pop();
        huffman_node* right_child = priority_queue_huffman_nodes.top(); priority_queue_huffman_nodes.pop();

        huffman_node* merged = new huffman_node{ '\0', left_child->symbol_frequency + right_child->symbol_frequency,
            left_child, right_child };
        priority_queue_huffman_nodes.push(merged);
    }

    huffman_node* root = priority_queue_huffman_nodes.top();//корень дерева хаффмана
    map<char, string> huffman_codes;//генерация кодов символов
    generate_huffman_codes(root, "", huffman_codes);

```

```
    map<char, string> huffman_codes;//генерация кодов символов
    generate_huffman_codes(root, "", huffman_codes);

    cout << "symbols codes:\n";//отображение кодов символов
    for (auto pair : huffman_codes) {
        cout << pair.first << " -> " << pair.second << endl;
    }

    string encoded_text = "";
    for (char c : input_text) {
        encoded_text += huffman_codes[c];
    }
    cout << "\nencoded text: " << encoded_text << endl;

    //сжатый текст
    int original_size_bits = input_text.size() * 8;//исходный размер в битах (по 8 бит на символ)
    int compressed_size_bits = encoded_text.size();//сжатый размер в битах

    //коэффициенты сжатия
    double compression_ratio_ascii = (double)original_size_bits / compressed_size_bits;
    double average_code_length = 0.0;
    double variance = 0.0;

    for (auto pair : huffman_codes) {//расчет средней длины кода и дисперсии
        int code_length = pair.second.size();
        average_code_length += code_length * (pair.second.size() / (double)input_text.size());
    }

    for (auto pair : huffman_codes) {//вычисление дисперсии длины кодов
        int code_length = pair.second.size();
        variance += (code_length - average_code_length) * (code_length - average_code_length) *
            (pair.second.size() / (double)input_text.size());
    }

    //коэффициент сжатия относительно равномерного кода
    int uniform_code_size = (int)ceil(log2(frequency_map.size())) * input_text.size();//равномерный код
    double compression_ratio_uniform = (double)original_size_bits / uniform_code_size;
```

```
103     //коэффициент сжатия относительно равномерного кода
104     int uniform_code_size = (int)ceil(log2(frequency_map.size())) * input_text.size();//равномерный код
105     double compression_ratio_uniform = (double)original_size_bits / uniform_code_size;
106
107     cout << "\ncompressed size in bits: " << compressed_size_bits << endl;
108     cout << "compression ratio (ASCII): " << compression_ratio_ascii << endl;
109     cout << "compression ratio (uniform code): " << compression_ratio_uniform << endl;
110     cout << "average code length: " << average_code_length << " bits\n";
111     cout << "variance code length: " << variance << endl;
112
113     string decoded_text = "";//декодирование текста
114     huffman_node* current_node = root;
115     for (char bit : encoded_text) {
116         if (bit == '0') {
117             current_node = current_node->left_child;
118         }
119         else {
120             current_node = current_node->right_child;
121         }
122
123         if (!current_node->left_child && !current_node->right_child) {
124             decoded_text += current_node->symbol;
125             current_node = root;
126         }
127     }
128
129     cout << "\ndecoded text: " << decoded_text << endl;
130
131     delete_huffman_tree(root);
132
133     return 0;
134 }
135
```

Тестирование:

```
symbols codes:
  -> 1011
D -> 11100
G -> 11101
V -> 11010
a -> 1111
c -> 0000
d -> 1001
h -> 1000
i -> 01
l -> 001
m -> 11011
n -> 0001
o -> 1010
r -> 11000
v -> 11001

encoded text: 11101101000110011011111001111000101010011011110100011111100101110110111000101011001010000 1000

compressed size in bits: 93
compression ratio (ASCII): 2.15054
compression ratio (uniform code): 2
average code length: 11 bits
variance code length: 112.24

decoded text: Gold Daniil Vladimirovich
```

Таблица частот:

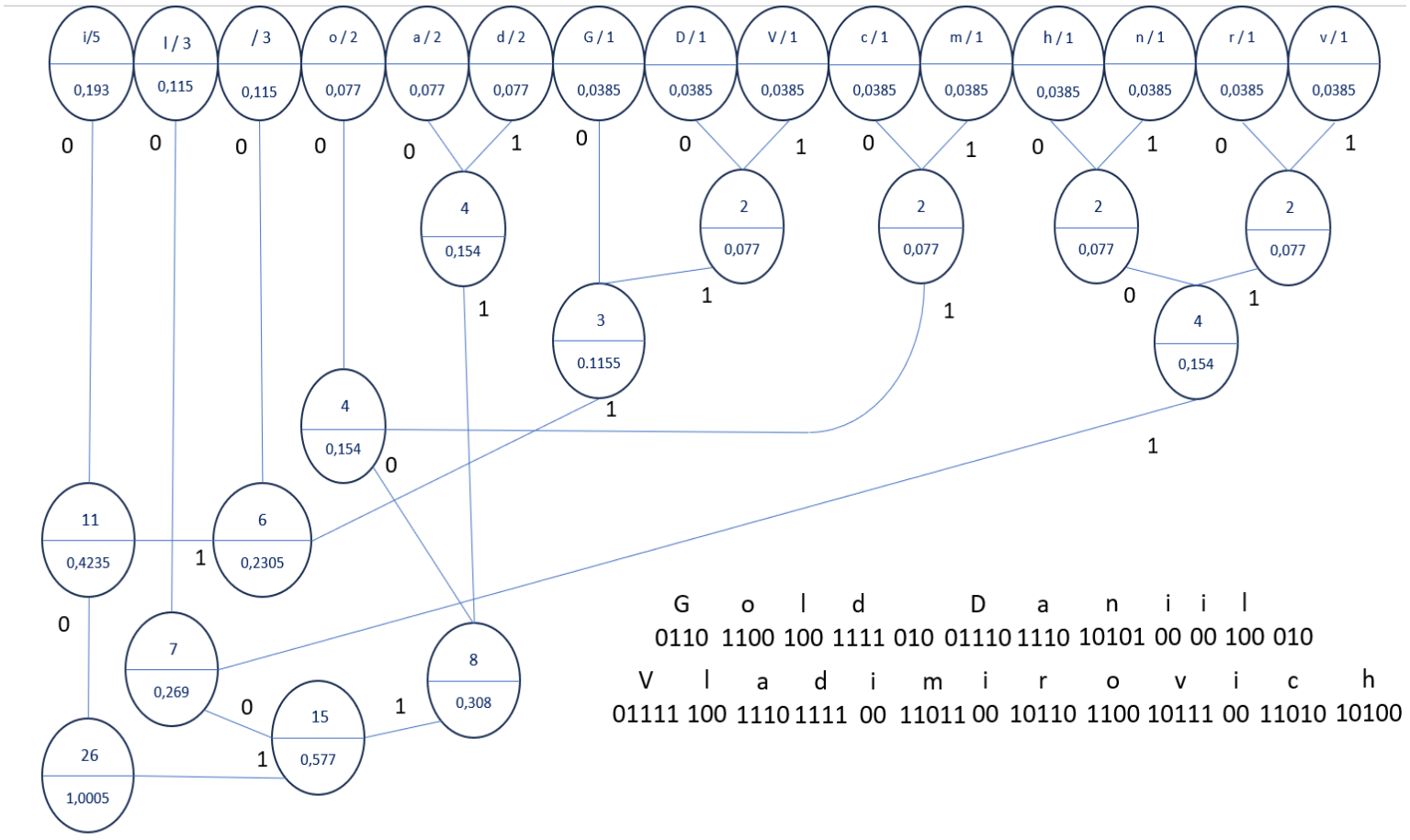
Алфавит	G	o	l	d		D	a	n	i	V
Кол.вх.	1	2	3	2	3	1	2	1	5	1
Вероятн.	0,0385	0,077	0,115	0,077	0,115	0,0385	0,077	0,0385	0,193	0,0385

Алфавит	m	r	v	c	h
Кол.вх.	1	1	1	1	1
Вероятн.	0,0385	0,0385	0,0385	0,0385	0,0385

Таблица отсортированных частот:

Алфавит	i	l		o	a	d	G	D	V	c
Кол.вх.	5	3	3	2	2	2	1	1	1	1
Вероятн.	0,193	0,115	0,115	0,077	0,077	0,077	0,0385	0,0385	0,0385	0,0385

Алфавит	m	h	n	r	v
Кол.вх.	1	1	1	1	1
Вероятн.	0,0385	0,0385	0,0385	0,0385	0,0385



Практическая оценка сложности алгоритма Хаффмана

Алгоритм Хаффмана состоит из нескольких этапов, каждый из которых имеет свою сложность:

- Подсчёт частоты символов: O(m)

Где m — длина исходного текста (проход по тексту для подсчёта частот).

- Построение дерева Хаффмана: O(nlog(n))

Где n — количество уникальных символов во входных данных

- Кодирование текста: O(m)

Замена каждого символа текста его кодом.

- Общая сложность: O(m+nlogn)

ВЫВОД

В ходе исследования были реализованы и протестированы алгоритмы сжатия текста: Шеннона-Фано, LZ77, LZ78 и Хаффмана.