Федеральное государственное бюджетное образовательное учреждение высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

Кафедра прикладной математики и кибернетики

Отчёт
по практической работе №2 «Побуквенное кодирование текстов»

Выполнил:
студент группы ИП-014
Бессонов А.О.

Работу проверил:
старший преподаватель
Дементьева К.И.

Новосибирск 2024 г.

Постановка задачи

Цель работы: Экспериментальное изучение избыточности сжатия текстового файла.

Задание:

1. Запрограммировать процедуру двоичного кодирования текстового файла побуквенным кодом. В качестве методов сжатия использовать метод Хаффмана и метод Шеннона (или метод Фано). Текстовые файлы использовать те же, что и в практической работе 1.

2. Вычислить среднюю длину кодовых слов и оценить избыточность кодирования для каждого построенного побуквенного кода.

3. После кодирования текстового файла вычислить оценки энтропии файла с закодированным текстом H_1, H_2, H_3 (после кодирования последовательность содержит 0 и 1) и заполнить таблицу.

Избыточность кодирования определяется как $r = L_{\text{ср}} - H$, где H – предельная энтропия текста, $L_{\text{ср}}$– средняя длина кодовых слов.

4. Оформить отчет, загрузить отчет в электронную среду. Отчет обязательно должен содержать заполненную таблицу и анализ полученных результатов. По желанию в отчет можно включить описание программной реализации. В отчет не нужно включать содержимое этого файла.

5. Анализ полученных результатов можно оформить в виде ответов на вопросы

Ход работы

Для первого файла был выбран алфавит:

Символы	a	b	c	d	e	f

Для второго файла был выбран следующий набор вероятностей символов:

Символы	a	b	c	d	e	f
Вероятности	0.36	0.181	0.179	0.12	0.09	0.07

Для третьего файла был выбран художественный текст на английском языке «Красавица и Чудовище».

Результаты работы программы:

```
FILE : equal_prob.txt
ENCODING NAME : Fano
        AVERAGE CODE LENGTH : 2.66382
        ENTROPY OF 1 : 0.990139
        ENTROPY OF 2 : 0.990051
        ENTROPY OF 3 : 0.989324
        CODING REDUNDANCY : 1.67368

FILE : equal_prob.txt
ENCODING NAME : Huffman
        AVERAGE CODE LENGTH : 2.66019
        ENTROPY OF 1 : 0.988776
        ENTROPY OF 2 : 0.988689
        ENTROPY OF 3 : 0.987748
        CODING REDUNDANCY : 1.67141
```

```
FILE : diff_prob.txt
ENCODING NAME : Fano
        AVERAGE CODE LENGTH : 2.43674
        ENTROPY OF 1 : 0.997175
        ENTROPY OF 2 : 0.990393
        ENTROPY OF 3 : 0.987908
        CODING REDUNDANCY : 1.43956

FILE : diff_prob.txt
ENCODING NAME : Huffman
        AVERAGE CODE LENGTH : 2.43362
        ENTROPY OF 1 : 0.978013
        ENTROPY OF 2 : 0.976991
        ENTROPY OF 3 : 0.976516
        CODING REDUNDANCY : 1.45561
```

```
FILE : Beauty_and_the_Beast.txt
ENCODING NAME : Fano
        AVERAGE CODE LENGTH : 4.1312
        ENTROPY OF 1 : 0.999641
        ENTROPY OF 2 : 0.998737
        ENTROPY OF 3 : 0.997665
        CODING REDUNDANCY : 3.13156

FILE : Beauty_and_the_Beast.txt
ENCODING NAME : Huffman
        AVERAGE CODE LENGTH : 4.11543
        ENTROPY OF 1 : 0.994005
        ENTROPY OF 2 : 0.993999
        ENTROPY OF 3 : 0.993277
        CODING REDUNDANCY : 3.12142
```

Метод кодирования	Название текста	Оценка избыточности кодирования	H_1	H_2	H_3
Фано	equal_prob.txt	1.67368	0.990139	0.990051	0.989324
Хаффман	equal_prob.txt	1.67141	0.988776	0.988689	0.987748
Фано	diff_prob.txt	1.43956	0.997175	0.990393	0.987908
Хаффман	diff_prob.txt	1.45561	0.978013	0.976991	0.976516
Фано	Beauty_and_the_Beast.txt	3.13156	0.999641	0.998737	0.997665
Хаффман	Beauty_and_the_Beast.txt	3.12142	0.994005	0.993999	0.993277

Выводы

В практической работе были реализованы 2 метода побуквенного кодирования: метод Хаффмана и метод Фано.

По результатам можно заметит, что в первом файле оценки избыточности максимально схожи. В двух файла из трех метод Хаффмана имеет более меньшую избыточность, чем метод Фано. Но во втором файле, где символы имеют весомое различие в вероятностях, избыточность кодирования меньше у Фано.

По структуре построения кодов метод Хаффмана и метод Фано довольно похожи, но метод Хаффман по моему предположению является более эффективным. Это связано с тем, что в методе Фано на начальных этапах вероятности могут быть разделены на множества не лучшим образом, в то время как Хаффман строится на каждом шаге учитывая все вероятности.

Сравнивая энтропию закодированных файлов, можно заметить, что во всех случая метод Хаффмана имеет меньшую энтропию. По моему мнению, по энтропии закодированных файлов можно судить о эффективности и качестве метода кодировки. Так как чем меньше мера неопределенности, тем предсказуемее закодированная последовательность, что скорее всего является следствием более лучшего сжатия.

Код программы

```cpp
#include <cmath>
#include <fstream>
#include <iostream>
#include <list>
#include <map>
#include <random>
#include <vector>
#include <utility>
#include <queue>

using namespace std;

char check_char(char c) {
    if (!isalpha(c) && !isdigit(c) && c != ' ')
        return -1;

    if (isalpha(c))
        return tolower(c);

    return c;
}

void check_probabilities(vector<double> probabilities) {
    double remains = 1;
    int index_of_max = 0;

    for (int i = 0; i < probabilities.size(); i++) {
        remains -= probabilities[i];
        if (probabilities[i] > probabilities[index_of_max])
            index_of_max = i;
    }

    probabilities[index_of_max] += remains;
}

void check_probabilities(vector<pair<char, double>> probabilities) {
    double remains = 1;
    int index_of_max = 0;

    for (int i = 0; i < probabilities.size(); i++) {
        remains -= probabilities[i].second;
        if (probabilities[i].second > probabilities[index_of_max].second)
            index_of_max = i;
    }

    probabilities[index_of_max].second += remains;
}

void generate_file_equal_prob(vector<char> alphabet) {
    ofstream file("equal_prob.txt");

    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<int> distribution(0, alphabet.size() - 1);

    for (long i = 0; i < 1 << 15; i++) {
        file << alphabet[distribution(gen)];
    }
```

```
    file.close();
}

void generate_file_diff_prob(vector<char> alphabet, vector<double>
alphabet_probabilities) {
    ofstream file("diff_prob.txt");

    random_device rd;
    mt19937 gen(rd());
    discrete_distribution<> distribution(alphabet_probabilities.begin(),
alphabet_probabilities.end());

    for (long i = 0; i < 1 << 15; i++) {
        file << alphabet[distribution(gen)];
    }

    file.close();
}

double calculate_entropy(vector<double> probabilities) {
    double entropy = 0;
    for (double probability : probabilities)
        entropy += probability * log2(probability);

    return -entropy;
}

double get_evaluation(string file_name, int limit) {
    if (limit < 1 || limit > 100) return -1;

    map<string, long> alphabet;
    list<char> buffer;
    long counter = 0;

    ifstream file(file_name);

    char c;
    while (file.get(c)) {
        if (check_char(c) != -1) {
            if (buffer.size() == limit) {
                buffer.pop_front();
            }

            buffer.push_back(check_char(c));

            if (buffer.size() == limit) {
                string str = "";
                for (char symbol : buffer)
                    str += symbol;

                alphabet[str]++;
                counter++;
            }
        }
    }

    file.close();

    vector<double> probabilities;
```

```
    for (const auto& symbol : alphabet) {
        probabilities.push_back((double)symbol.second / counter);
        //cout << symbol.first << " " << symbol.second << "\n";
    }

    check_probabilities(probabilities);

    return calculate_entropy(probabilities) / limit;
}

vector<pair<char, double>> calculate_probabilities(string file_name) {
    map<char, long> alphabet;
    long counter = 0;

    ifstream file(file_name);

    char c;
    while (file.get(c)) {
        if (check_char(c) != -1) {
            alphabet[check_char(c)]++;
            counter++;
        }
    }

    file.close();

    vector<pair<char, double>> probabilities;

    for (const auto& symbol : alphabet) {
        probabilities.push_back(make_pair(symbol.first,
(double)symbol.second / counter));
        //cout << symbol.first << " " << symbol.second << "\n";
    }

    check_probabilities(probabilities);

    return probabilities;
}

double calculate_average_code_length(map<char, string> codes, map<char,
double> probabilities) {
    double length = 0;
    for (const auto& code : codes)
        length += code.second.size() * probabilities.at(code.first);
    return length;
}

void encode_file(string file_name, string encoding_name, map<char, string>
codes, map<char, double> probabilities) {
    ifstream input(file_name);
    ofstream output("encoded_" + encoding_name + "_" + file_name);

    char c;
    while (input.get(c)) {
        if (check_char(c) != -1) {
            output << codes.at(check_char(c));
        }
    }
```

```
    input.close();
    output.close();
}

void evaluate_file_encoding(string file_name, string encoding_name,
map<char, string> codes, map<char, double> probabilities) {
    double avg_length = calculate_average_code_length(codes, probabilities);

    double max_entropy = -1;
    double entropy[3];
    for (int i = 0; i < 3; i++) {
        entropy[i] = get_evaluation("encoded_" + encoding_name + "_" +
file_name, i + 1);
        max_entropy = max(max_entropy, entropy[i]);
    }

    cout << "FILE : " << file_name << "\n";
    cout << "ENCODING NAME : " << encoding_name << "\n";
    cout << "\tAVERAGE CODE LENGTH : " << avg_length << "\n";

    for (int i = 0; i < 3; i++) {
        cout << "\tENTROPY OF " << i + 1 << " : " << entropy[i] << "\n";
        //cout << "\t\tCODING REDUNDANCY : " << avg_length - entropy[i] <<
"\n";
    }

    cout << "\tCODING REDUNDANCY : " << avg_length - max_entropy << "\n\n";

    /*
    for (const auto& code : codes)
        cout << code.first << " " << code.second << " | " <<
probabilities[code.first] << "\n";
    cout << "\n";
    */
}

struct Huffman_node {
    char symbol;
    double probability;
    Huffman_node* left;
    Huffman_node* right;

    Huffman_node(char symbol, double probability) : symbol(symbol),
probability(probability), left(nullptr), right(nullptr) {}
};

struct Compare_nodes {
    bool operator()(Huffman_node* lhs, Huffman_node* rhs) const {
        return lhs->probability > rhs->probability;
    }
};

Huffman_node* build_Huffman_tree(vector<pair<char, double>>& probabilities)
{
    priority_queue<Huffman_node*, vector<Huffman_node*>, Compare_nodes>
node_queue;

    for (const auto& probability : probabilities) {
        node_queue.push(new Huffman_node(probability.first,
probability.second));
```

```
    }

    while (node_queue.size() > 1) {
        Huffman_node* left_node = node_queue.top();
        node_queue.pop();

        Huffman_node* right_node = node_queue.top();
        node_queue.pop();

        Huffman_node* merged_node = new Huffman_node('\0', left_node-
>probability + right_node->probability);
        merged_node->left = left_node;
        merged_node->right = right_node;

        node_queue.push(merged_node);
    }

    return node_queue.top();
}

void build_Huffman_codes(Huffman_node* root, const string& code, map<char,
string>& huffman_codes) {
    if (root) {
        if (!root->left && !root->right) {
            huffman_codes[root->symbol] = code;
            return;
        }

        build_Huffman_codes(root->left, code + "0", huffman_codes);
        build_Huffman_codes(root->right, code + "1", huffman_codes);
    }
}

void encode_Huffman_file(string file_name) {
    vector<pair<char, double>> probabilities =
calculate_probabilities(file_name);
    map<char, string> huffman_codes;

    Huffman_node* root = build_Huffman_tree(probabilities);
    build_Huffman_codes(root, "", huffman_codes);

    map<char, double> probabilities_map;
    for (const auto& probability : probabilities)
        probabilities_map[probability.first] = probability.second;

    encode_file(file_name, "Huffman", huffman_codes, probabilities_map);
    evaluate_file_encoding(file_name, "Huffman", huffman_codes,
probabilities_map);
}

void build_Fano_codes(vector<pair<char, double>>& probabilities, int start,
int end, string code, map<char, string>& fano_codes) {
    if (start == end) {
        fano_codes[probabilities[start].first] = code;
        return;
    }

    int mid = start;
    double min_delta = 10;
    double left_sum = 0;
```

```
    double right_sum = 0;

    for (int i = start; i <= end; i++)
        right_sum += probabilities[i].second;

    for (int i = start; i <= end; i++) {
        left_sum += probabilities[i].second;
        right_sum -= probabilities[i].second;

        if (abs(left_sum - right_sum) < min_delta) {
            mid = i;
            min_delta = abs(left_sum - right_sum);
        }
    }

    build_Fano_codes(probabilities, start, mid, code + '0', fano_codes);
    build_Fano_codes(probabilities, mid + 1, end, code + '1', fano_codes);
}

void encode_Fano_file(string file_name) {
    vector<pair<char, double>> probabilities =
calculate_probabilities(file_name);
    sort(probabilities.begin(), probabilities.end(),
        [](const auto& left, const auto& right) { return left.second >
right.second; });

    map<char, string> fano_codes;
    build_Fano_codes(probabilities, 0, probabilities.size() - 1, "",
fano_codes);

    map<char, double> probabilities_map;
    for (const auto& probability : probabilities)
        probabilities_map[probability.first] = probability.second;

    encode_file(file_name, "Fano", fano_codes, probabilities_map);
    evaluate_file_encoding(file_name, "Fano", fano_codes,
probabilities_map);
}

int main() {
    vector<char> alphabet = { 'a', 'b', 'c', 'd', 'e', 'f'};
    vector<double> alphabet_probabilities = { 0.36, 0.181, 0.179, 0.12,
0.09, 0.07 };

    if (alphabet.size() != alphabet_probabilities.size()) {
        cout << "ERROR";
        return -1;
    }
    check_probabilities(alphabet_probabilities);

    generate_file_equal_prob(alphabet);
    generate_file_diff_prob(alphabet, alphabet_probabilities);

    vector<string> files = { "equal_prob.txt" , "diff_prob.txt" ,
"Beauty_and_the_Beast.txt" };
    for (string file : files) {
        encode_Fano_file(file);
        encode_Huffman_file(file);
    }
```

```
    return 0;
}
```