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

Кафедра прикладной математики и кибернетики

Отчёт
по практической работе №3 «Блочное кодирование»

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

Работу проверил:
старший преподаватель
Дементьева К.И.

Новосибирск 2024 г.

Постановка задачи

Цель работы: Экспериментальное изучение свойств блочного кодирования.

Задание:

1. Для выполнения работы необходим сгенерированный файл с неравномерным распределением из практической работы 1.

При блочном кодировании входная последовательность разбивается на блоки равной длины, которые кодируются целиком. Поскольку вероятностное распределение символов в файле известно, то и вероятности блоков могут быть вычислены и использованы для построения кода.

2. Закодировать файл блочным методом кодирования (можно использовать любой метод кодирования), размер блока n = 1,2,3,4 . Вычислить избыточность кодирования на символ входной последовательности для каждого размера блока.

3. После тестирования программы необходимо заполнить таблицу и проанализировать полученные результаты, сравнить с теоретическими оценками.

Ход работы

Для файла был выбран следующий набор вероятностей символов:

Символы	a	b
Вероятности	0.25	0.75

Результаты работы программы:

```
THEORETICAL ENTROPY : 0.811278
PRACTICAL ENTROPY   : 0.808688

BLOCK SIZE : 1  | ENCODING NAME : Huffman
        AVERAGE CODE LENGTH           : 1
        THEORETICAL CODING REDUNDANCY : 0.188722
        PRACTICAL CODING REDUNDANCY   : 0.191312

BLOCK SIZE : 2  | ENCODING NAME : Huffman
        AVERAGE CODE LENGTH           : 0.84375
        THEORETICAL CODING REDUNDANCY : 0.0324719
        PRACTICAL CODING REDUNDANCY   : 0.0350618

BLOCK SIZE : 3  | ENCODING NAME : Huffman
        AVERAGE CODE LENGTH           : 0.822917
        THEORETICAL CODING REDUNDANCY : 0.0116385
        PRACTICAL CODING REDUNDANCY   : 0.0142284

BLOCK SIZE : 4  | ENCODING NAME : Huffman
        AVERAGE CODE LENGTH           : 0.818359
        THEORETICAL CODING REDUNDANCY : 0.00708125
        PRACTICAL CODING REDUNDANCY   : 0.00967115
```

	Длина блока n=1	Длина блока n=2	Длина блока n=3	Длина блока n=4
Оценка избыточности кодирования на один символ входной последовательности	0.191312	0.0350618	0.0142284	0.00967115

Выводы

В практической работе был реализован метод блочного кодирования, где полученные блоки кодируются с помощью метода Хаффмана. С помощью разных длин блоков можно улучшить эффективность сжатия. В процессе расчета вероятностей для блоков полученные значения значительно различаются, так что для оптимального кодирования блоков используется метод Хаффмана.

По результатам оценок избыточности кодирования на один символ входной последовательности при увеличении длины блока можно заметить, что избыточность уменьшается. Это связано с тем, что вероятности встречи различных последовательности начинают иметь большое влияние на кодирование, поэтому при росте длины этой последовательности уменьшается избыточность кодирования.

Код программы

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

using namespace std;

double theoretical_entropy = 0;
double practical_entropy = 0;

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<string, 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_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 < 3 * (1 << 14); 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<string, double>> calculate_block_probabilities(vector<char>
alphabet, vector<double> alphabet_probabilities, int block_size) {
    vector<pair<string, double>> probabilities;
    int block_alphabet_size = 1;

    for (int i = 0; i < block_size; i++)
        block_alphabet_size *= alphabet.size();

    for (int i = 0; i < block_alphabet_size; i++) {
        string block_string = "";
        double block_probability = 1;

        int block = i;
        for (int j = 0; j < block_size; j++) {
            block_string += alphabet[block % alphabet.size()];
            block_probability *= alphabet_probabilities[block %
alphabet.size()];
            block /= alphabet.size();
        }

        probabilities.push_back(make_pair(block_string, block_probability));
    }

    check_probabilities(probabilities);

    return probabilities;
}

double calculate_average_code_length(map<string, string> codes, map<string,
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, int block_size,
map<string, string> codes, map<string, double> probabilities) {
    ifstream input(file_name);
    ofstream output("encoded_" + encoding_name + "_blocksize_" +
to_string(block_size) + "_" + file_name);

    char c;
    string buffer = "";
    while (input.get(c)) {
        if (check_char(c) != -1) {
            buffer += check_char(c);

            if (buffer.length() == block_size) {
                output << codes.at(buffer);
                buffer = "";
            }
        }
    }

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

```
void evaluate_file_encoding(string file_name, string encoding_name, int
block_size, map<string, string> codes, map<string, double> probabilities) {
    double avg_length = calculate_average_code_length(codes, probabilities)
/ block_size;

    cout << "BLOCK SIZE : " << block_size << "\t| ENCODING NAME : " <<
encoding_name << "\n";
    cout << "\tAVERAGE CODE LENGTH            : " << avg_length << "\n";
    cout << "\tTHEORETICAL CODING REDUNDANCY : " << avg_length -
theoretical_entropy << "\n";
    cout << "\tPRACTICAL CODING REDUNDANCY   : " << avg_length -
practical_entropy << "\n\n";

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

    cout << "\n";
    */
}

struct Huffman_node {
    string data;
    double probability;
    Huffman_node* left;
    Huffman_node* right;

    Huffman_node(string data, double probability) : data(data),
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<string, 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<string,
string>& huffman_codes) {
    if (root) {
        if (!root->left && !root->right) {
            huffman_codes[root->data] = 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, int block_size,
vector<pair<string, double>> probabilities) {
    map<string, string> huffman_codes;

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

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

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

void build_Fano_codes(vector<pair<string, double>>& probabilities, int
start, int end, string code, map<string, 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, int block_size, vector<pair<string,
double>> probabilities) {
    sort(probabilities.begin(), probabilities.end(),
        [](const auto& left, const auto& right) { return left.second >
right.second; });

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

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

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

    vector<char> alphabet = { 'a', 'b'};
    vector<double> alphabet_probabilities = { 0.25, 0.75 };

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

    generate_file_diff_prob(alphabet, alphabet_probabilities);

    theoretical_entropy = calculate_entropy(alphabet_probabilities);
    practical_entropy = get_evaluation("diff_prob.txt", 1);

    cout << "THEORETICAL ENTROPY : " << theoretical_entropy << "\n";
    cout << "PRACTICAL ENTROPY   : " << practical_entropy << "\n\n";

    for (int block_size = 1; block_size <= 4; block_size++) {
        vector<pair<string, double>> probabilities =
calculate_block_probabilities(alphabet, alphabet_probabilities, block_size);
        //encode_Fano_file("diff_prob.txt", block_size, probabilities);
        encode_Huffman_file("diff_prob.txt", block_size, probabilities);
    }

    return 0;
}
```