

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

КУРСОВАЯ РАБОТА
по дисциплине «Алгоритмы и Структуры Данных»
Тема: Кодирование и декодирование методами Хаффмана и Фано-
Шеннона. Исследование

Студент гр. 8304

Воропаев А.О.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2019

ЗАДАНИЕ НА КУРСОВУЮ РАБОТУ

Студент Воропаев Антон Олегович

Группа 8304

Тема работы: Кодирование и декодирование методами Хаффмана и Фано-Шеннона

Содержание пояснительной записки:

- Содержание
- Введение
- Алгоритм Хаффмана
- Алгоритм Фано-Шеннона
- Тестирование
- Исходный код

Дата выдачи задания: 11.10.2019

Дата сдачи реферата:

Дата защиты реферата: 17.12.2019

Студент

Воропаев А.О.

Преподаватель

Фирсов М.А.

АННОТАЦИЯ

В данной работе была создана программа на языке программирования C++, которая сочетает в себе функции ввода исходной строки, её кодировки и декодировки алгоритмами Хаффмана и Фано-Шэннона. Были использованы преимущества C++ для минимизации кода. Для понимания результата исследования в коде были приведены отладочные выводы.

SUMMARY

In this work, a program was created in the C ++ programming language, which combines the input function of the source string, its encoding and decoding by Huffman and Fano-Shannon algorithms. The benefits of C ++ were used to minimize code. To understand the result of the research, debugging conclusions were given in the code.

СОДЕРЖАНИЕ

Введение	5
1. Алгоритм Хаффмана	6
2. Алгоритм Фано-Шэннона	6
3. Функции и структуры данных	7
4. Тестирование	9

Введение

Целью данной курсовой работы является реализация алгоритмов кодирования/декодирования Хаффмана и Фано-Шэннона. Также необходимо определить качество сжатия, время работы алгоритмов и сделать соответствующие выводы об их эффективности .

1. Алгоритм Хаффмана

Для решения поставленной подзадачи была написана функция `HF_tree`, которая строит дерево по алгоритму Хаффмана. Изначально во входной строке подсчитывается кол-во повторений каждого символа. При построении дерева на каждой итерации выбираются два символа с наименьшим кол-вом повторений во входной строке, для них создаётся новый элемент, который будет их родителем в дереве. Между элементами устанавливается связь, при этом элемент с большим кол-вом повторений помещается в левую ветку, а с наименьшим в правую. Затем по построенному дереву составляется словарь, в котором каждому символу сопоставляется его код, для последующей кодировки. Декодирование также происходит по построенному ранее дереву без помощи словаря.

2. Алгоритм Фано-Шеннона

Для решения поставленной подзадачи была написана рекурсивная функция `FS_tree`, которая строит дерево по алгоритму Фано-Шэннона. Изначально во входной строке подсчитывается кол-во повторений каждого символа. При построении дерева общее кол-во повторений символов делится примерно поровну для каждого шага рекурсии, также на каждом шаге элемент с большим кол-вом повторений помещается в левую ветку, а с наименьшим в правую. Затем по построенному дереву составляется словарь, в котором каждому символу сопоставляется его код, для последующей кодировки. Декодирование также происходит по построенному ранее дереву без помощи словаря.

Итог:

Скорость работы обоих алгоритмов имеет разницу только на очень больших объемах данных. Алгоритм Хаффмана является усовершенствованной версией алгоритма Фано-Шэннона из-за чего длина закодированного им

сообщения всегда меньше либо равна длине строки после применения алгоритма Фано-Шэннона. Это происходит из-за не всегда оптимального разделения символов при шаге алгоритма Фано-Шэннона. В ходе исследования было замечено, что чем больше различных символов необходимо закодировать, тем больше становится разница в длине закодированных сообщений.

3. ФУНКЦИИ И СТРУКТУРЫ ДАННЫХ

1. Структуры

```
struct Node {  
    Node() = default;  
  
    int cnt = 0;  
    std::string s;  
    std::shared_ptr<Node> left;  
    std::shared_ptr<Node> right;  
};
```

Структура Node представляет узел дерева, где left – указатель на левую ветку, right – на правую, s – строка символов содержащихся в узле, cnt – счетчик повторений этих символов в исходном тексте.

```
struct char_count{  
    char_count(int count, std::string &s){  
        cnt = count;  
        str = s;  
    };  
  
    int cnt;  
    std::string str;  
};
```

Структура char_count необходима для подсчета кол-ва повторений символов в строке.

2. Функции

1. **void** read_file(std::vector<std::string>& file_data, std::ifstream& input)

Данная функция предназначена для считывания входных из файлового потока ввода данных и записи их в вектор `file_data`.

2. `bool` decode(std::string const& code, std::shared_ptr<Node>& head, std::string& result)

Данная функция предназначена для декодирования сообщения.

`code` – закодированное сообщение

`head` – корень построенного дерева

`result` – строка, в которую запишется результат декодирования

3. `bool` comparator (char_count i, char_count j)

Компаратор для сортировки вектора структур типа `Node` по убыванию кол-ва повторений в тексте

4. `void` count_chars(std::string &input_str, std::vector<char_count> &cnt_vector)

Функция для подсчёта кол-ва повторений каждого символа в исходной строке и их сортировки.

5. `void` HF_tree(std::string &input_str, std::vector<char_count> &cnt_vector, std::shared_ptr<Node> &head)

Функция для построения дерева по алгоритму Хаффмана.

`input_str` – входная строка

`cnt_vector` – отсортированный по убыванию кол-ва повторений каждого символа в исходном тексте вектор.

`head` – указатель, в который будет записан адрес корня сформированного дерева.

6. `void` make_dict(std::shared_ptr<Node> &elem, std::map<char, std::string> &dict, std::string ¤t_code)

Рекурсивная функция для построения словаря, в котором каждому символу сопоставляется его код для последующей шифровки.

`elem` – обрабатываемый узел дерева

dict – словарь, в который записываются данные

current_code – текущий код, к которому на каждом шаге добавляется 0 либо 1 в зависимости от того к какой ветке мы переходим.

7. void encode(std::string input_str, std::map<char, std::string> &dict, std::string &encode_str)

Функция для декодирования зашифрованной строки.

input_str – входная строка

dict – словарь, в котором каждому символу сопоставлен его код.

encode_str – закодированная строка

8. void make_FS_tree(std::shared_ptr<Node> &elem, std::vector<char_count> cnt_vector)

Рекурсивная функция для построения дерева по алгоритму Фано-Шэннона.

elem – обрабатываемый узел дерева

cnt_vector – отсортированный по убыванию кол-ва повторений каждого символа в исходном тексте вектор.

9. void FS_tree(std::string &input_str, std::vector<char_count> &cnt_vector, std::shared_ptr<Node> &head)

Функция, формирующая корень дерева для алгоритма Фано-Шэннона и запускающая рекурсивную функцию для построения дерева.

4. ТЕСТИРОВАНИЕ

```
Input message: !@*$(^*)98636418599
Codes of characters for Haffman's algorithm
!=1011 | $=1001 | (=1000 | )=0110 | *=000 | 1=0011 | 3=0101 | 4=0100 | 5=0010 | 6=1110 | 8=1111 | 9=110 | @=1010 | ^=0111 |
Codes of characters for Fano-Shannon's algorithm
!=1010 | $=011111 | (=011110 | )=0110 | *=1110 | 1=0010 | 3=010 | 4=0011 | 5=000 | 6=1011 | 8=110 | 9=1111 | @=100 | ^=01110 |
Haffman's algorithm run time: 0ms
Fano-Shannon's algorithm run time: 0ms
Haffman's string has 6.57895%(5 characters) better compression
Haffman's algorithm: 1011101000010011000011100001101101111110010111100100001111110010110110
Fano-Shannon's algorithm: 1010100111001111101111001110111011111010110101011001100101100001111111
Result of decoding for two algorithms: !@*$(^*)98636418599
```

Рисунок 1 – Результат работы программы

В данной таблице представлены только результаты кодировки исходных строк, в самой программе присутствуют отладочные выводы, которые помогают лучше оценить два данных алгоритма.

INPUT	OUTPUT
!@*\$(^*)98636 418599	Haffman's algorithm: 1011101000010011000011100001101101111110010111100100 001111110010110110 Fano-Shannon's algorithm: 10101001110011111011110011101110011011111101011010101 10011001011000011111111
aaaaaaaaaaaaa	Haffman's algorithm: 1111111111111 Fano-Shannon's algorithm: 1111111111111
bababababa	Haffman's algorithm: 1010101010 Fano-Shannon's algorithm: 1010101010
qwertyuiop	Haffman's algorithm: 101101100011010001000111111011011100 Fano-Shannon's algorithm: 1111111111111011101101011000111011001000
!@#\$\$%^&*	Haffman's algorithm: 111110101100011010001000 Fano-Shannon's algorithm: 111111111011101101001101000
abbcccdddd	Haffman's algorithm: 1101111111010100000 Fano-Shannon's algorithm: 0001011010101111111

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы была написана программа, содержащая в себе реализацию кодирования и декодирования методом Хаффмана и Фано-Шэннона. Был получен опыт работы с дополнительными возможностями C++ и эффективной алгоритмизацией на нем. Также были закреплены знания полученные на протяжении семестра. Исходный код программы находится в приложении А.

ПРИЛОЖЕНИЕ А

КОД ПРОГРАММЫ

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <map>
#include <regex>
#include <ctime>

struct Node {
    Node() = default;

    int cnt = 0;
    std::string s;
    std::shared_ptr<Node> left;
    std::shared_ptr<Node> right;
};

struct char_count{
    char_count(int count, std::string &s){
        cnt = count;
        str = s;
    };

    int cnt;
    std::string str;
};

void read_file(std::vector<std::string>& file_data, std::ifstream& input)
{
    std::string current_file_string;
    while (std::getline(input, current_file_string))
    {
        if (current_file_string.back() == '\r')
            current_file_string.erase(current_file_string.end() - 1);
        file_data.push_back(current_file_string);
    }
}

bool decode(std::string const& code, std::shared_ptr<Node>& head, std::string&
result){

    std::string current_checked_code;
    std::shared_ptr<Node> processing_node(head);

    for(char c : code) {

        if(c == '0') {
            if(processing_node->right == nullptr) {
                return false;
            }
            if(processing_node->right->s.length() == 1) {
                result += processing_node->right->s;
                processing_node = head;
            }
            else
                processing_node = processing_node->right;
        }

        else{
```

```

        if(processing_node->left == nullptr) {
            return false;
        }
        if(processing_node->left->s.length() == 1) {
            result += processing_node->left->s;
            processing_node = head;
        }
        else
            processing_node = processing_node->left;
    }
}
return true;
}

bool comparator (char_count i, char_count j) {
    return (i.cnt > j.cnt);
}

void count_chars(std::string &input_str, std::vector<char_count> &cnt_vector) {
    for(auto &i : input_str){
        bool flag = false;
        for( auto &j : cnt_vector) {
            if(j.str[0] == i) {
                j.cnt += 1;
                flag = true;
                break;
            }
        }

        if(flag)
            continue;

        std::string tmp;
        tmp.push_back(i);
        char_count elem(1, tmp);
        cnt_vector.push_back(elem);
    }
    std::sort(cnt_vector.begin(), cnt_vector.end(), comparator);
}

void HF_tree(std::string &input_str, std::vector<char_count> &cnt_vector,
std::shared_ptr<Node> &head){

    //Create leaf_vector(vector of pointers of type Node);
    std::vector<std::shared_ptr<Node>> leaf_vector;
    for( auto &i : cnt_vector){

        std::shared_ptr<Node> leaf_ptr(new(Node));
        leaf_ptr->cnt = i.cnt;
        leaf_ptr->s = i.str;

        leaf_vector.push_back(leaf_ptr);
    }

    //Create tree
    while(leaf_vector.size() > 1){
        size_t i = leaf_vector.size() - 1;

        //creating of new Node out of two leaves; creating of connection
        std::shared_ptr<Node> leaf_ptr(new(Node));
        leaf_ptr->s = leaf_vector[i-1]->s + leaf_vector[i]->s;
    }
}

```

```

leaf_ptr->cnt = leaf_vector[i-1]->cnt + leaf_vector[i]->cnt;

if(leaf_vector[i]->cnt <= leaf_vector[i-1]->cnt) {
    leaf_ptr->right = leaf_vector[i];
    leaf_ptr->left = leaf_vector[i - 1];
}
else {
    leaf_ptr->left = leaf_vector[i];
    leaf_ptr->right = leaf_vector[i - 1];
}

//insert new Node at the right place in the vector
for(int j = 0; j < leaf_vector.size(); ++j){
    if(leaf_ptr->cnt >= leaf_vector[j]->cnt) {
        leaf_vector.insert(leaf_vector.begin() + j, leaf_ptr);
        break;
    }
}

leaf_vector.pop_back();
leaf_vector.pop_back();
}
head = leaf_vector[0];
}

void make_dict(std::shared_ptr<Node> &elem, std::map<char, std::string> &dict,
std::string &current_code){

    if(!elem->left && !elem->right){
        if(current_code.empty()) {
            current_code += '1';
        }
        dict[elem->s[0]] = current_code;
        return;
    }

    std::string left_code = current_code + '1';
    make_dict(elem->left, dict, left_code);

    std::string right_code = current_code + '0';
    make_dict(elem->right, dict, right_code);
}

void encode(std::string input_str, std::map<char, std::string> &dict,
std::string &encode_str){
    for(auto &i : input_str){
        encode_str += dict[i];
    }
}

void make_FS_tree(std::shared_ptr<Node> &elem, std::vector<char_count>
cnt_vector) {

    //Condition of end of recursion
    if(cnt_vector.size() == 1) {
        return;
    }

    //Create child Nodes and vectors for them
    std::shared_ptr<Node> right_branch(new Node);

```

```

std::shared_ptr<Node> left_branch(new Node);
std::vector<char_count> right_cnt_vector;
std::vector<char_count> left_cnt_vector;

//Set connection
elem->right = right_branch;
elem->left = left_branch;

//Creating of right branch and vector
for(size_t i = cnt_vector.size() - 1; i >= 1; --i) {

    if(right_branch->cnt + cnt_vector[i+1].cnt >= elem->cnt / 2) {
        break;
    }

    right_branch->cnt += cnt_vector[i].cnt;
    right_branch->s.insert(0, cnt_vector[i].str);
    right_cnt_vector.insert(right_cnt_vector.begin(), cnt_vector.back());
    cnt_vector.pop_back();
}

//Creating of left branch and vector
left_cnt_vector = cnt_vector;
left_branch->s = elem->s;
left_branch->s.erase(left_branch->s.end() - right_branch->s.length(),
left_branch->s.end());
left_branch->cnt = elem->cnt - right_branch->cnt;

//Recursive call
make_FS_tree(left_branch, left_cnt_vector);
make_FS_tree(right_branch, right_cnt_vector);
}

void FS_tree(std::string &input_str, std::vector<char_count> &cnt_vector,
std::shared_ptr<Node> &head) {

    int sum = 0;
    std::string all_chars;
    for(auto &i : cnt_vector) {
        sum += i.cnt;
        all_chars += i.str;
    }
    head->cnt = sum;
    head->s = all_chars;

    make_FS_tree(head, cnt_vector);
}

int main(int argc, char* argv[]) {

    if (argc >= 2)
    {
        std::ifstream input(argv[1]);
        if (!input.is_open())
        {
            std::cout << "Incorrect input file\n";
            return 1;
        }

        std::vector<std::string> file_data;

```

```

read_file(file_data, input);

for (int i = 0; i != file_data.size(); ++i)
{
    std::string input_str = file_data[i];
    if(input_str.empty()) {
        std::cout << "Nothing have been written into the string" <<
std::endl;
        return 1;
    }

    std::cout << "Test #" << i + 1 <<
    "\nInput message: " << input_str << std::endl;

    //Count number of repetition of every character in the input string
    std::vector<char_count> cnt_vector;
    count_chars(input_str, cnt_vector);

    double start_time = clock();
    //Haffman algorithm
    //Make tree
    std::shared_ptr<Node> hf_head(new Node);
    HF_tree(input_str, cnt_vector, hf_head);
    //Make dict
    std::map<char, std::string> hf_dict;
    std::string hf_proc_str;
    make_dict(hf_head, hf_dict, hf_proc_str);
    //Encode input string
    std::string hf_encode_str;
    encode(input_str, hf_dict, hf_encode_str);
    //Decode coded string
    std::string hf_decode_res;
    decode(hf_encode_str, hf_head, hf_decode_res);
    //Find Haffman's algorithm run time
    double end_time = clock();
    double hf_search_time = end_time - start_time;

    start_time = clock();
    //Fano-Shannon algorithm
    //Make tree
    std::shared_ptr<Node> fs_head(new Node);
    FS_tree(input_str, cnt_vector, fs_head);
    //Make dict
    std::map<char, std::string> fs_dict;
    std::string fs_proc_str;
    make_dict(fs_head, fs_dict, fs_proc_str);
    //Encode input string
    std::string fs_encode_str;
    encode(input_str, fs_dict, fs_encode_str);
    //Decode coded string
    std::string fs_decode_res;
    decode(fs_encode_str, fs_head, fs_decode_res);
    //Find Fano-Shannon's algorithm run time
    end_time = clock();
    double fs_search_time = end_time - start_time;

    std::cout << "Codes of characters for Haffman's algorithm" <<
std::endl;

```



```

        for(auto &j : hf_dict) {
            std::cout << j.first << '=' << j.second << " | ";
        }
        std::cout << std::endl << "Codes of characters for Fano-Shannon's
algorithm" << std::endl;
        for(auto &j : fs_dict) {
            std::cout << j.first << '=' << j.second << " | ";
        }
        std::cout << std::endl;

        std::cout << "Haffman's algorithm run time: " << hf_search_time <<
"ms" << std::endl;
        std::cout << "Fano-Shannon's algorithm run time: " <<
fs_search_time << "ms" << std::endl;

        std::cout << "Haffman's string has " <<
            static_cast<double>(fs_encode_str.length() -
hf_encode_str.length()) / fs_encode_str.length() * 100
            << "%" << fs_encode_str.length() -
hf_encode_str.length()
            << " characters) better compression" << std::endl;

        std::cout << "Haffman's algorithm: " << hf_encode_str << std::endl;
        std::cout << "Fano-Shannon's algorithm: " << fs_encode_str <<
std::endl;

        std::cout << "Result of decoding for two algorithms: " <<
fs_decode_res <<
            std::endl << "_____ "
<< std::endl;

    }
}
return 0;
}

```