МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

Институт Информационных технологий

Кафедра Математического обеспечения и стандартизации информационных технологий

ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ № 6

по дисциплине

«Структуры и алгоритмы обработки данных»

Тема: «Алгоритмы сжатия и кодирования данных»

Выполнила студентка группы ИКБО-01-19 Ославская Л.Я.

Принял преподаватель Грушицын А.С.

Практическая работа выполнена «__»_____202__ г. *(подпись студента)*

«Зачтено» «__»_____202__ г. *(подпись руководителя)*

Москва 2020

1. **Цель работы**

Изучение алгоритма оптимального префиксного кодирования Хаффмана. Практическое применение алгоритма Хаффмана для сжатия данных

2. **Постановка задачи**

Провести кодирование исходной строки символов «ославская лидия ярославовна» с использованием алгоритма Хаффмана. Исходная строка символов, таким образом, определяет индивидуальный вариант задания для каждого студента. Исходную строку брать в нижнем регистре.

Для выполнения работы необходимо выполнить следующие действия:

1. Построить таблицу частот встречаемости символов в исходной строке символов для чего сформировать алфавит исходной строки и посчитать количество вхождений (частот) символов и их вероятности появления, например, для строки **пупкин василий кириллович** такая таблица будет иметь вид (на шестой позиции в таблице находится пробел):

Алфавит	п	у	к	и	н	« »	в
Кол. вх.	2	1	2	6	1	2	2
Вероятн.	0.08	0.04	0.08	0.24	0.04	0.08	0.08
Алфавит	а	с	л	й	р	о	ч
Кол. вх.	1	1	3	1	1	1	1
Вероятн.	0.04	0.04	0.12	0.04	0.04	0.04	0.04

Рис. 1 Пример таблицы частоты встречаемости символов

2. Отсортировать алфавит в порядке убывания частот появления символов.

Алфавит	и	л	п	к	« »	в	у
Кол. вх.	6	3	2	2	2	2	1
Вероятн.	0.24	0.12	0.08	0.08	0.08	0.08	0.04
Алфавит	н	а	с	й	р	о	ч
Кол. вх.	1	1	1	1	1	1	1
Вероятн.	0.04	0.04	0.04	0.04	0.04	0.04	0.04

Рис. 2 Пример отсортированной таблицы частоты встречаемости символов

3. Построить дерево кодирования Хаффмана.

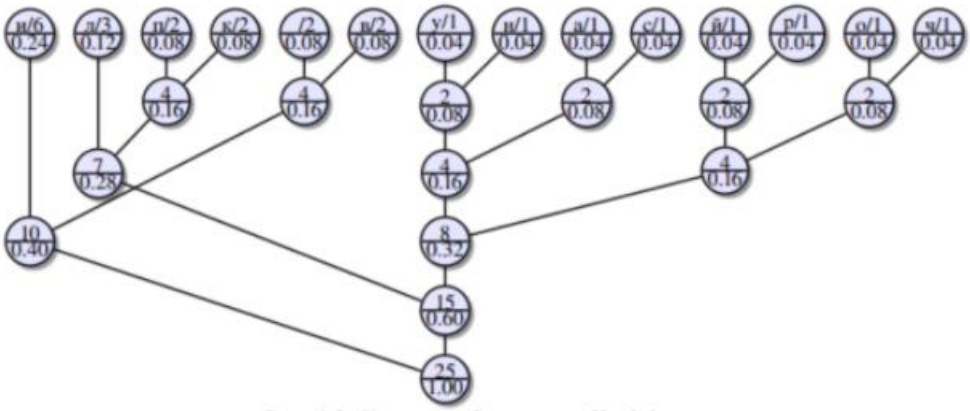

Рис. 3 Пример дерева кодирования Хаффмана

4. Упорядочить построенное дерево слева-направо (при необходимости). Присвоить ветвям коды. Определить коды символов.

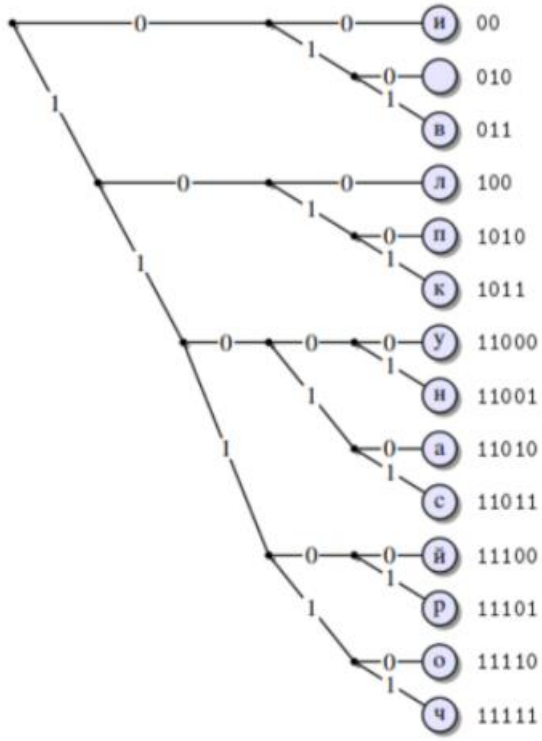

Рис. 4 Пример упорядоченного дерева кодирования Хаффмана

5. Провести кодирование исходной строки.

```
 п    у    п    к   и    н   « »  в    а    с   и   л   и    й
1010 11000 1010 1011 00 11001 010 011 11010 11011 00 100 00 11100
« »   к   и    р   и   л   л    о   в   и    ч
010 1011 00 11101 00 100 100 11110 011 00 11111
```

Рис. 5 Пример кодирования исходной строки

Рассчитать коэффициенты сжатия относительно кодировки ASCII и относительно равномерного кода.

6. Рассчитать среднюю длину полученного кода и его дисперсию.

Сделать выводы о проделанной работе, основанные на полученных результатах.

Оформить отчет с подробным описанием хода решения задачи, описанием текста исходного кода и проведенного тестирования программы.

3. Решение

Код Хаффмана - под кодированием символов из некоторого множества будем понимать установление соответствия каждому из них некоторой битовой последовательности. Мы уже давно знакомы с некоторыми кодировки. Например с 7-битной ASCII кодировкой или 8-битной Windows-1251. Это так называемые равномерные кодировки. Это означает, что длина кодирующей последовательности одинакова для всех символов. Идея кодирования Хаффмана состоит в отказе от равномерности кода — символы, которые встречаются в кодируемом тексте чаще предполагается кодировать более короткими битовыми последовательностями. Такие коды в которых длина кодов различных символов отличается, называют неравномерными.

Алгоритм Хаффмана является примером жадного алгоритма в котором строится бинарное дерево листьями которого являются символы текста. Глубина листа (расстояние до корня) тем меньше, чем выше встречаемость символа в тексте. Бинарное дерево кода строится путем последовательного объединения наиболее редких символов. При этом образуется новая вершина, соответствующая множеству объединяемых символов. Новой вершине приписывается частота равная сумме частот объединяемых вершин. Процесс объединения продолжается до получения дерева. Т.е. в списке множеств символов останется только один элемент — объединение всех символов текста. Частота этого последнего узла равна количеству символов в тексте.

Реализация алгоритма Хаффмана:

```
struct Node{
	char ch;
	int freq;
	Node* left, * right;
```

```
};
```

4. Тестирование

Производим тестирование реализации алгоритма Хаффмана. На рисунке 6 у нас показана таблица частоты встречаемости символов, в первом столбике символы, во втором количество этих символов в строке, в третьем вероятность появления символов. На рисунке 7 показана сортированная по вероятностям, таблица частоты встречаемости символом. На рисунке 8 мы показываем построенное дерево Хаффмана. На рисунке 9 мы выводим, сортированное по убыванию, дерево Хаффмана. На рисунке 10 представлена кодированная алгоритмом Хаффмана строка. На рисунке 11 декодируем наш код Хаффмана. На рисунке 12 выводим на экран подсчеты средней длины и дисперсии, а также коэффициенты сжатия относительно ASCII и относительно равномерного кода.

Таблица вероятностей

Символ	Количество	Вероятность
о	3	0.111111
с	3	0.111111
л	3	0.111111
а	4	0.148148
	2	0.0740741
к	1	0.037037
в	3	0.111111
я	3	0.111111
и	2	0.0740741
д	1	0.037037
р	1	0.037037
н	1	0.037037

Рис. 6 Таблица частоты встречаемости символов

```
Сортированная таблица вероятностей
Символ  Количество      Вероятность
а       4               0.148148
о       3               0.111111
с       3               0.111111
л       3               0.111111
в       3               0.111111
я       3               0.111111
        2               0.0740741
и       2               0.0740741
к       1               0.037037
д       1               0.037037
р       1               0.037037
н       1               0.037037
```

Рис. 7 Сортированная таблица частоты встречаемости символов

```
Дерево Хаффмана
л-000
я-001
к-10100
в-010
с-011
о-100
н-10101
д-10110
р-10111
а-111
 -1100
и-1101
```

Рис. 8 Дерево Хаффмана

```
Сортированное Дерево Хаффмана
р-10111
д-10110
н-10101
к-10100
и-1101
 -1100
а-111
о-100
с-011
в-010
я-001
л-000
```

Рис. 9 Сортированное дерево Хаффмана

```
Кодированная строка :
1000110001110100111010011100111000001101101101101001110000110111100011000111010100010101011111
```

Рис. 10 Кодированная строка

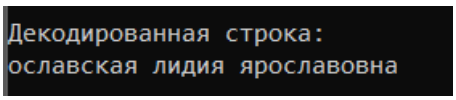

Рис. 11 Декодированная строка

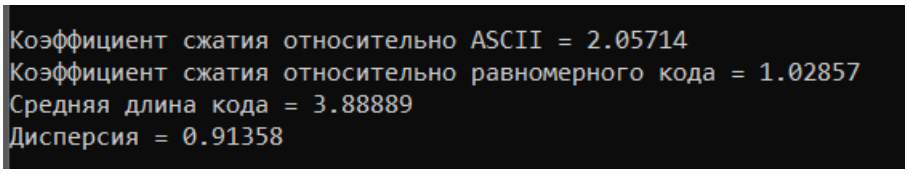

Рис. 12 Вывод коэффициентов сжатия, средней длины и дисперсии

Из результатов программы видно:

1. Коэффициент сжатия относительно ASCII равен 2.05714, а коэффициент относительно равномерного кода равен 1.02857.
2. Средняя длина кода равна 3.88889, а дисперсия 0.91358.
3. Для наиболее часто встречающихся символов назначается код меньшего размера, а по мере уменьшения встречаемости символов в тексте их код будет более длинным.
4. Чем чаще символ встречается в строке, тем большая у него вероятность.

5. Вывод

В результате выполнения работы я:

1. Освоила алгоритм Хаффмана. А также его реализацию на языке программирования C++.
2. Научилась реализовывать процедуры для работы с алгоритмом Хаффмана.

6. Исходный код программы

```
#include <iostream>
#include <string>
#include <queue>
#include <algorithm>
#include <unordered_map>
```

```
using namespace std;

// Узел дерева
struct Node
{
	char ch;
	int freq;
	Node* left, * right;
};

// Функция для выделения нового узла дерева
Node* getNode(char ch, int freq, Node* left, Node* right)
{
	Node* node = new Node();
	node->ch = ch;
	node->freq = freq;
	node->left = left;
	node->right = right;
	return node;
}
// Объект сравнения, который будет использоваться для упорядочивания кучи
struct comporator
{
	bool operator()(Node* l, Node* r)
	{
		return l->freq > r->freq;
	}
};
// пройти по дереву Хаффмана и сохранить коды Хаффмана в map
void Encode(Node* root, string str,
	unordered_map<char, string>& huffmanCode)
{
	if (root == nullptr)
		return;
	if (!root->left && !root->right) {
		huffmanCode[root->ch] = str;
	}
	Encode(root->left, str + "0", huffmanCode);
	Encode(root->right, str + "1", huffmanCode);
}
// проходим по дереву Хаффмана и декодируем закодированную строку
void Decode(Node* root, int& index, string str)
{
	if (root == nullptr) {
		return;
	}
	if (!root->left && !root->right)
	{
		cout << root->ch;
		return;
	}
	index++;
	if (str[index] == '0')
		Decode(root->left, index, str);
	else
		Decode(root->right, index, str);
}

bool cmp(pair<char, int>& a,
	pair<char, int>& b)
{
	return a.second > b.second;
}
bool cmp2(pair<char, string>& a,
	pair<char, string>& b)
```

```
{
	return a.second > b.second;
}


// Строим дерево Хаффмана
void HuffmanTree(string text)
{
	double sr = 0;
	vector<double> probability;
	vector<double> kol;
	vector<char> symbols;
	double ver = 0;
	int counter = 0;
	double len = text.length();
	// подсчитываем частоту появления каждого символа и сохраняем в freq
	unordered_map<char, int> freq;
	for (char ch : text) {
		freq[ch]++;
	}
	cout << "	Таблица вероятностей" << endl;
	cout << "Символ" << "\t" << "Количество" << "\t" << "Вероятность" << endl;
	//выводим таблицу вероятностей
	for (auto it = freq.begin(); it != freq.end(); ++it)
	{
		symbols.push_back((char)it->first);
		counter++;
		ver = (double)it->second / len;
		probability.push_back(ver);
		kol.push_back(it->second);
		cout<< it->first << "\t" << it->second <<"\t\t" <<ver<< endl;
	}
	cout << endl;
	cout << "Сортированная таблица вероятностей" << endl;
	cout << "Символ" << "\t" << "Количество" << "\t" << "Вероятность" << endl;
	vector<pair<char, int> > A; //создаем вектор для сортирвоки
	for (auto& it : freq) { //с map все передаем в вектор
		A.push_back(it);
	}
	sort(A.begin(), A.end(), cmp); //сортируем
	for (auto& it : A) {
		ver = (double)it.second / len;
		cout << it.first << "\t"
			<< it.second << "\t\t" << ver << endl;
	}
	// Создаем приоритетную очередь для хранения активных узлов
	priority_queue<Node*, vector<Node*>, comporator> pq;
	// Создаем листовой узел для каждого символа и добавляем его в приоритетную
очередь pq
	for (auto pair : freq) {
		pq.push(getNode(pair.first, pair.second, nullptr, nullptr));
	}
	// делаем, пока в очереди не будет более одного узла
	while (pq.size() != 1)
	{
		// Удаляем два узла с наивысшим приоритетом
		Node* left = pq.top(); pq.pop();
		Node* right = pq.top();	pq.pop();
		// Создаем новый внутренний узел с этими двумя узлами
		int sum = left->freq + right->freq;
		//Добавляем новый узел в приоритетную очередь.
		pq.push(getNode('\0', sum, left, right));
	}
	// корень хранит указатель на корень дерева Хаффмана
	Node* root = pq.top();
```

```
// пройти по дереву Хаффмана и сохранить коды Хаффмана в huffmanCode
unordered_map<char, string> huffmanCode;
Encode(root, "", huffmanCode);
vector<double> bit;
cout << endl;
//вывод дерева Хаффмана
cout << "Дерево Хаффмана\n";
for (auto pair : huffmanCode) {
	cout << pair.first << "-" << pair.second << '\n';
	bit.push_back(pair.second.size());
}
cout << endl;
cout << "Сортированное Дерево Хаффмана\n";
vector<pair<char, string> > B; //создаем вектор для сортирвоки
for (auto& it : huffmanCode) { //с map все передаем в вектор
	B.push_back(it);
}
sort(B.begin(), B.end(), cmp2); //сортируем
vector<pair<char, string> > v1;
vector<pair<char, string> > v2;
vector<pair<char, string> > v3;
for (auto& it : B) {
	if (it.second.size() > 4)
	{
		v1.push_back(it);
	}
	else if (it.second.size() > 3)
	{
		v2.push_back(it);
	}
	else if (it.second.size() > 2)
	{
		v3.push_back(it);
	}
}
for (auto& it : v1)//выводим сортированное дерево
{
	cout << it.first << "-" << it.second << endl;
}
for (auto& it : v2)
{
	cout << it.first << "-" << it.second << endl;
}
for (auto& it : v3)
{
	cout << it.first << "-" << it.second << endl;
}
double ascii = 8 * len;
double huff = 0;
double res = 0;
double res2 = 0;
//считаем коэффициенты сжатия относительно кодировки ASCII
for (int i = 0; i < counter; i++) {
	huff += kol[i] * bit[i];
}
for (int i = 0; i < counter; i++) {
	res = ascii / huff;
}
cout << endl;
cout << "Коэффициент сжатия относительно ASCII = " << res<<endl;
//считаем коэффициенты сжатия относительно равномерного кода
for (int i = 0; i < counter; i++) {
	res2 = len * ceil(log2(counter)) / huff;
}
cout << "Коэффициент сжатия относительно равномерного кода = " << res2 << endl;
```

```
	//считаем среднюю длину полученного кода
	for (int i = 0; i < counter; i++) {
		sr += probability[i] * bit[i];
	}
	cout << "Средняя длина кода = " << sr << endl;
	//считаем дисперсию
	double dis = 0;
	for (int i = 0; i < counter; i++) {
		double k = (bit[i] - sr);
		dis += probability[i] * pow(k, 2);
	}
	cout << "Дисперсия = " << dis<<endl;

	string str = "";
	for (char ch : text) {
		str += huffmanCode[ch];
	}
	cout << "\nКодированная строка :\n" << str << '\n';
	int index = -1;
	cout << "\nДекодированная строка: \n";
	while (index < (int)str.size() - 2) {
		Decode(root, index, str);
	}

	sort(probability.begin(), probability.end(), greater<double>());
	sort(kol.begin(), kol.end(), greater<double>());
	cout << endl;

}

int main()
{
	setlocale(LC_ALL, "RU");
	string text = "ославская лидия ярославовна";
	HuffmanTree(text);
	return 0;
}
```