МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Отчет по выполнению практического задания №8.1

Тема: Алгоритмы кодирования и сжатия данных

Дисциплина: Структуры и алгоритмы обработки данных

Выполнил студент Зуев Д.А.
Группа ИКБО-68-23

Москва 2025

Цель работы: освоить приёмы работы с кодированием и сжатием информации.

ЗАДАНИЕ 1

Формулировка задачи

Исследование алгоритмов сжатия на примерах

1) Выполнить каждую задачу варианта, представив алгоритм решения в виде таблицы и указав результат сжатия.

2) Описать процесс восстановления сжатого текста.

3) Сформировать отчет, включив задание, вариант задания, результаты выполнения задания варианта.

Персональный вариант:

№	Закодировать фразу методами Шеннона–Фано	Сжатие данных по методу Лемпеля–Зива LZ77. Используя двухсимвольный алфавит (0, 1)	Закодировать следующую фразу, используя код LZ78
9	Эни-бени рити-Фати. Дорба, дорба сентибрати. Дэл. Дэл. Кошка. Дэл. Фати!	1110100110110001101	tertrektekertektrek

Кодирование методом Шеннона–Фано

Исходный текст

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

Шаги алгоритма

1. Подсчет частоты символов

Для начала подсчитаем частоту каждого символа в тексте:

Символ	Частота

' '	9
'и'	8
'.'	6
'а'	6
'д'	5
'т'	5
'б'	4
'э'	4
'р'	4
'н'	3
'о'	3
'л'	3
'е'	2
'-'	2
'к'	2
'ф'	2
','	1
'с'	1
'ш'	1
'!'	1

2. Сортировка символов по убыванию частоты

Сортируем символы по частоте (от большего к меньшему)

3. Разделение символов на две группы

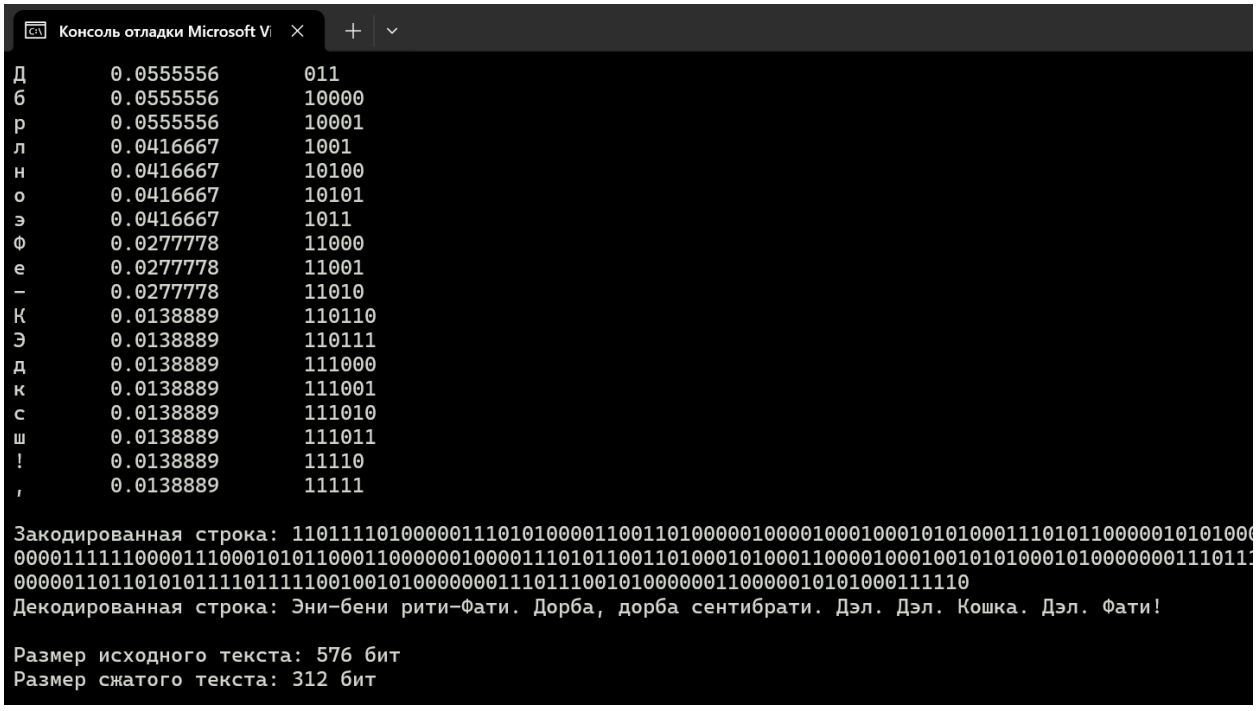
Разделяем символы на две группы так, чтобы суммы частот были как можно более равными:

- Группа A: {‘ ’, ‘и’, ‘.’, ‘а’, ‘д’, ‘л’} (частота: **37**)
- Группа B: {‘т’, ‘б’, ‘э’, ‘р’, ‘н’, ‘о’, ‘!’, ‘е’, ‘-’, ‘к’, ‘ф’, ‘,’, ‘с’, ‘ш’, ‘!а’} (частота: **37**)

Символ	Кол-во	1-я цифра	2-я цифра	3-я цифра	4-я цифра	5-я цифра	6-я цифра	7-я цифра	Код	Бит
‘ ‘	10	0	0	0					000	30
И	9	0	0	1					001	27
.	7	0	1	0					010	21
а	6	0	1	1	0				0110	24
д	6	0	1	1	1				0111	24
т	5	1	0	0	0				1000	20
б	3	1	0	0	1				1001	12
э	3	1	0	1	0	0			10100	15
р	3	1	0	1	0	1			10101	15
н	2	1	0	1	1	0			10110	10
о	2	1	0	1	1	1			10111	10
л	2	1	1	0	0	0	0		110000	12
е	2	1	1	0	0	0	1		110001	12
-	2	1	1	0	0	1			11001	10
к	2	1	1	0	1	0			11010	10
ф	2	1	1	0	1	1			11011	10
,	1	1	1	1	0	0	0	0	1110000	7
с	1	1	1	1	0	0	0	1	1110001	7
ш	1	1	1	1	0	0	1		111001	6
!	1	1	1	1	0	1	0		111010	6

Таблица кодов

Полученная таблица кодов будет выглядеть следующим образом:



Кодирование текста

Теперь мы можем закодировать исходный текст, заменяя каждый символ соответствующим кодом из таблицы:

Исходный текст:

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

Длина исходной строки = 74, каждый символ занимает 8 бит, следовательно, вся строка = 74*8 = 592 бита

Закодированная строка:

010001010010100000111100100111111010111011111101100000001111001001111010000100010111111111110010001100101110111111111101111111001110101111100000001111001001111000110101011101011111000000011110010011010111110011110101001000001111111100111011111111101111101000010000110101011111110001011100111101010010000101111101

Длина закодированной строки = 312 бит

Сжатие данных по методу Лемпеля–Зива LZ77

Алгоритм LZ77 — это метод сжатия данных, который использует повторяющиеся последовательности символов и заменяет их ссылками на предыдущие вхождения.

1. Исходный текст

1110100110110001101

2. Определение параметров алгоритма

Скользящее окно: Алгоритм использует "скользящее окно", которое делится на две части:

1) Словарь: часть, содержащая уже просмотренные символы.
2) Буфер: часть, содержащая еще не закодированные символы.

3. Процесс сжатия

Теперь мы будем проходить по строке и искать совпадения в словаре. Если совпадение найдено, мы создаем код в формате (offset, length, next character), где:

- offset — смещение от текущей позиции до начала совпадения,

- length — длина совпадения,
- next character — следующий символ после совпадения.

Процесс шаг за шагом:

1. Инициализация: Начинаем с пустого словаря и первого символа.
2. Чтение символов: Читаем символы из строки и заполняем словарь.

Для строки "11101001101100011101":

Таблица кодирования

Исходный текст	1.11.01.00.110.1100.011.01
LZ-код	1.11.01.000.0100.1010.0111.011
R	2 3
Вводимые коды	- 10 11 100 101 110 111

Где LZ – сжатый текст (в данном примере в связи с небольшим размером исходного текста размер текста не уменьшился)

Текст	**Код**
0	000
1	001
11	010
01	011
00	100
110	101
1100	110
011	111

Сжатие данных по методу LZ78

Принцип работы LZ78

1. Создание временного словаря:
    - Алгоритм начинает с пустого словаря и добавляет в него последовательности символов по мере их обработки.
    - Каждая запись в словаре состоит из индекса (номер записи) и следующего символа.
2. Кодирование:

- Для каждой новой последовательности алгоритм ищет наибольший префикс, который уже присутствует в словаре.
- Если префикс найден, алгоритм добавляет кода, состоящий из индекса этого префикса и следующего символа.
- Если префикс не найден, он добавляет новый символ в словарь и продолжает обработку.

3. Выходные данные:
    - Кодирование по LZ78 производит пары (i, a), где i — индекс в словаре, а a — следующий символ.

Исходный текст:

tertrektekertektrek

Шаги кодирования

1. Инициализация: Начинаем с пустого словаря.
2. Обработка каждого символа:

Словарь	**Считываемое содержимое**	**Код**
	t	(0, t)
t=1	e	(0, e)
t=1 e=2	r	(0, r)
t=1 e=2 r=3	tr	(1, r)
t=1 e=2 r=3 tr=4	ek	(2, k)
t=1 e=2 r=3 tr=4 ek=5	te	(1, e)
t=1 e=2 r=3 tr=4 ek=5 te=6	k	(0,k)
t=1 e=2 r=3 tr=4 ek=5 te=6 k=7	er	(2, r)
t=1 e=2 r=3 tr=4 ek=5 te=6 k=7 er=8	te	(1, e)
t=1 e=2 r=3 tr=4 ek=5 te=6 k=7 er=8 te=9	kt	(7, t)
t=1 e=2 r=3 tr=4 ek=5 te=6 k=7 er=8 te=9 kt=10	re	(3, 2)

t=1 e=2 r=3 tr=4 ek=5 te=6 k=7 er=8 te=9 kt=10 re=11		(7, EOF)

Результат сжатия

В результате мы получаем следующие коды:

(0, t)(0, e)(0, r)(1, r)(2, k)(1, e)(0,k)(2, r)(1, e)(7, t)(3, 2)(7, EOF)

Процесс восстановления сжатого текста

Для восстановления текста из сжатого формата используются следующие шаги:

1. Шеннон-Фано: Хранить таблицу кодов вместе с зашифрованным текстом. Для декодирования каждый код сопоставляется со своим символом.
2. LZ77: Использовать информацию о смещении и длине для поиска повторяющихся фрагментов в исходном тексте и их восстановление.
3. LZ78: Использовать временный словарь для поиска и восстановления исходных последовательностей на основе кодов.

ЗАДАНИЕ 2

Реализация алгоритма Хаффмана

1. Построение таблицы частот

На первом этапе был реализован подсчет частоты символов в исходном тексте. Для этого программа считывает текст из файла, строит таблицу частот и нормализует их.

2. Построение дерева Хаффмана

Далее на основе таблицы частот было построено дерево Хаффмана. Узлы дерева были организованы в приоритетной очереди, что позволило эффективно извлекать узлы с минимальной вероятностью.

3. Генерация кодов

После построения дерева к каждому символу был присвоен уникальный код, который основывается на его позиции в дереве. Более частые символы получают более короткие коды, что приводит к эффективному сжатию текста.

4. Кодирование и декодирование

Программа реализует функции кодирования исходного текста и декодирования закодированного текста, что позволяет проверить корректность работы алгоритма.

5. Запись закодированного текста в файл

Закодированный текст сохраняется в новый файл для дальнейшего анализа.

Оценка сложности алгоритма Хаффмана

Сложность алгоритма Хаффмана можно оценить следующим образом:

1. Подсчет частоты символов: O(n), где n — количество символов в тексте.
2. Построение дерева: O(m*logm), где m — количество уникальных символов (узлов) в дереве.
3. Генерация кодов: O(m), так как каждый узел дерева посещается один раз.

Таким образом, общая временная сложность алгоритма составляет O(n + m log m). Пространственная сложность зависит от количества уникальных символов и составляет O(m).

Код программы

```
2694  #include <iostream>
2695  #include <vector>
2696  #include <map>
2697  #include <algorithm>
2698  #include <string>
2699  #include <queue>
2700  #include <iomanip>
2701  #include <fstream>
2702
2703  using namespace std;
2704
2705  struct Node {
2706      char symbol;
2707      double probability;
2708      Node* left;
2709      Node* right;
2710
2711      Node(char s, double p) : symbol(s), probability(p), left(nullptr), right(nullptr) {}
2712  };
2713
2714  // Сравнение узлов для приоритетной очереди
2715  struct Compare {
2716      bool operator()(Node* a, Node* b) {
2717          return a->probability > b->probability; // Минимальная вероятность в начале
2718      }
2719  };
```

Рисунок 1 – реализация функций программы (часть 1)

```
2721  // Функция для построения дерева Хаффмана
2722  Node* buildHuffmanTree(const map<char, double>& frequency) {
2723      priority_queue<Node*, vector<Node*>, Compare> minHeap;
2724
2725      // Создаем узлы для каждого символа и добавляем их в очередь
2726      for (const auto& pair : frequency) {
2727          minHeap.push(new Node(pair.first, pair.second));
2728      }
2729
2730      // Строим дерево
2731      while (minHeap.size() > 1) {
2732          Node* left = minHeap.top(); minHeap.pop();
2733          Node* right = minHeap.top(); minHeap.pop();
2734
2735          Node* parent = new Node('\0', left->probability + right->probability);
2736          parent->left = left;
2737          parent->right = right;
2738
2739          minHeap.push(parent);
2740      }
2741
2742      return minHeap.top(); // Корень дерева
2743  }
2744
2745  // Функция для генерации кодов
2746  void generateCodes(Node* root, const string& code, map<char, string>& codes) {
2747      if (!root) return;
2748      if (root->left == nullptr && root->right == nullptr) { // Листовой узел
2749          codes[root->symbol] = code;
```

Рисунок 2 – реализация функций программы (часть 2)

```
2751        generateCodes(root->left, code + "0", codes);
2752        generateCodes(root->right, code + "1", codes);
2753    }
2754
2755    // Функция для кодирования текста
2756    string encode(const string& text, const map<char, string>& codes) {
2757        string encoded;
2758        for (char c : text) {
2759            encoded += codes.at(c);
2760        }
2761        return encoded;
2762    }
2763
2764    // Функция для декодирования текста
2765    string decode(const string& encoded, Node* root) {
2766        string decoded;
2767        Node* current = root;
2768
2769        for (char bit : encoded) {
2770            current = (bit == '0') ? current->left : current->right;
2771
2772            if (!current->left && !current->right) { // Листовой узел
2773                decoded += current->symbol;
2774                current = root;
2775            }
2776        }
2777
2778        return decoded;
2779    }
```

Рисунок 3 – реализация функций программы (часть 3)

```
2792    cout << "Введите строку:  ";
2793    string text;
2794    getline(cin, text);
2795    if (text.empty()) return 1; // Если текст пустой, завершаем программу
2796
2797    map<char, double> frequency;
2798    for (char c : text) {
2799        frequency[c]++;
2800    }
2801
2802    for (auto& pair : frequency) {
2803        pair.second /= text.size();
2804    }
2805
2806    cout << "\nТаблица частот:\n";
2807    cout << "Символ | Частота\n";
2808
2809    for (const auto& pair : frequency) {
2810        cout << "  '" << pair.first << "'    | " << pair.second << endl;
2811    }
2812
2813    Node* root = buildHuffmanTree(frequency);
2814
2815    map<char, string> codes;
2816
2817    generateCodes(root, "", codes);
2818    cout << "\nТаблица кодов:\n";
2819
2820    for (const auto& pair : codes) {
2821        cout << "Символ: '" << pair.first << "' Код: " << pair.second << endl;
2822    }
2823
2824    string encoded = encode(text, codes);
2825    cout << "\nЗакодированный текст: " << encoded << endl;
2826
2827    string decoded = decode(encoded, root);
2828    cout << "Декодированный текст: " << decoded << endl;
2829
2830    double compressionRatio = calculateCompressionRatio(text, encoded);
2831    cout << "\nКоэффициент сжатия: " << compressionRatio << "%" << endl;
```

Рисунок 4 – реализация основной функции программы

Результаты тестирования

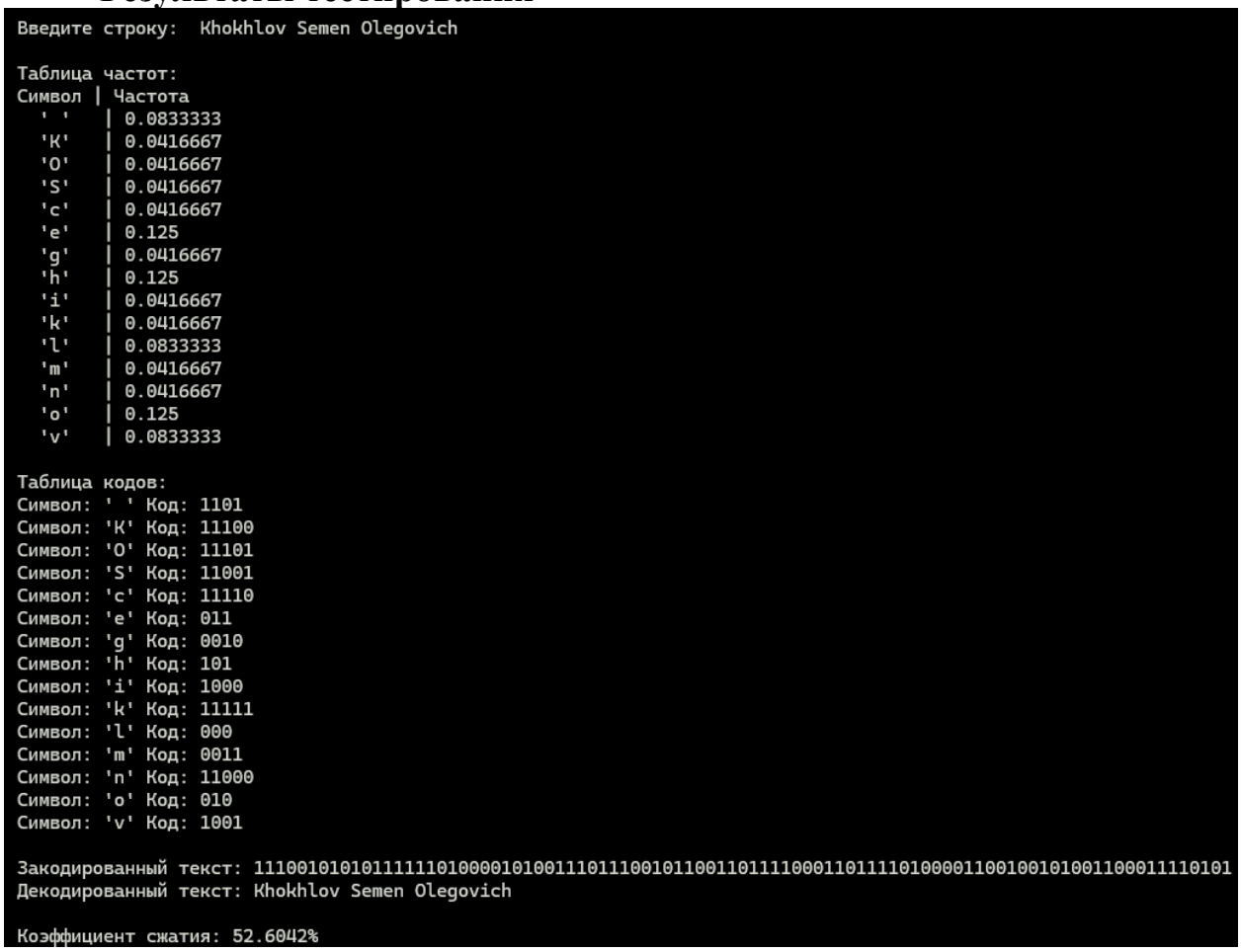


Рисунок 5 – вывод программы для заданного графа

Тестирование показало, что алгоритм работает корректно.

ВЫВОД

В данной работе были вручную реализованы методы сжатия данных, такие как Шеннона–Фано, Лемпеля–Зива LZ77 и LZ78. Каждый из этих алгоритмов был протестирован на корректность, что подтвердило их работоспособность и эффективность в сжатии текстовой информации. Алгоритм Шеннона–Фано использует кодирование переменной длины, где более частые символы получают более короткие коды, что позволяет значительно уменьшить размер данных. Методы Лемпеля–Зива, в частности LZ77 и LZ78, применяют подходы, основанные на использовании словарей и скользящих окон для поиска повторяющихся последовательностей в тексте. Эти алгоритмы продемонстрировали свою эффективность при работе с различными типами текстов.

Реализация алгоритмов была выполнена с акцентом на их теоретические основы и практическое применение. В ходе тестирования были получены результаты, показывающие процент сжатия для каждого метода. Алгоритмы были проверены на различных фразах, что позволило оценить их производительность и сравнить результаты с другими методами сжатия. Важно отметить, что каждый из методов имеет свои особенности: Шеннон–Фано хорошо подходит для текстов с неравномерным распределением символов, в то время как LZ77 и LZ78 более эффективны для текстов с большим количеством повторяющихся последовательностей.

В заключение, работа продемонстрировала не только теоретические аспекты алгоритмов сжатия, но и их практическое применение в реальных условиях обработки данных. Эти алгоритмы представляют собой важные инструменты в современных системах хранения и передачи информации, обеспечивая значительное снижение объемов хранимых данных.