Московский авиационный институт
(национальный исследовательский университет)
Институт № 8 «Информационные технологии и прикладная математика»

Курсовой проект
по курсу «Дискретный анализ»
«Алгоритм LZW»

Студент: Минеева Светлана Алексеевна

Группа: М8О-310Б-21

Преподаватель: Сорокин С.А.

Оценка: ____

Дата: 17.11.2023

Подпись: _____

Москва, 2023

Содержание:

1. Постановка задачи

Реализуйте алгоритм LZW.

Начальный словарь выглядит следующим образом: a -> 0 b -> 1 c -> 2 ... x -> 23 y -> 24 z -> 25 EOF -> 26

Формат ввода:

Вам будут даны тесты двух типов. Первый тип: compress <text>

Текст состоит только из малых латинских букв. В ответ на него вам нужно вывести коды, которыми будет закодирован данный текст.

Второй тип: decompress <codes>

Вам даны коды в которые был сжат текст из малых латинских букв, вам нужно его разжать.

Формат вывода:

В ответ на тест первого типа вам нужно вывести коды, которыми будет закодирован данный текст через пробел.

В ответ на тест второго типа выведите разжатый текст.

2. Общий метод и алгоритм решения

LZW (Lempel-Ziv-Welch) - это алгоритм сжатия данных без потерь на основе словаря, который был разработан в 1984 году. Он предназначен для уменьшения объема данных, представляя повторяющиеся последовательности битов более компактно. Особенность и преимущество алгоритма – легкая реализация.

Рассмотрим процесс сжатия данных. Инициализируется словарь, в него помещается алфавит рассматриваемого языка. С входного потока считывается строка символов и посимвольным проходом происходит проверка, существует ли в таблице такая строка. Если есть, то считываем следующий символ, иначе добавляем код для предыдущей найденной строки в результат, строку заносим в таблицу и продолжаем поиск до окончания входного текста. Данный процесс можно визуализировать следующей последовательностью шагов:

Шаг 1. Символы алфавита рассматриваемого языка заносим в словарь. Входная строка A пополняется первым символом текста.

Шаг 2. Считываем следующий символ текста (вид анализируемой строки – AB).

Шаг 3. Если считанный только что считанный символ – это окончание текста, то в результат добавляется код для A, иначе если строка AB есть в словаре, то A = AB и далее шаг 2, а если AB нет в словаре, то в результат добавляем код для строки A, строку AB добавляем в словарь, A = B и далее шаг 2.

Декодирование происходит аналогичным способом. При получении нового кода создается новая строка в словаре, а при получении существующего кода в результат из словаря добавляется строка с рассматриваемым кодом.

3. Текст программы

```
#include <iostream>
#include <vector>
#include <unordered_map>

struct TrieNode {
    std::unordered_map<char, TrieNode*> children;
    int index;

    TrieNode() : index(-1) {}
};

class Trie {
    public:
        Trie() {
            root = new TrieNode();
            next_index = 0;
        }

        void Insert(const std::string& line) {
            TrieNode* node = root;

            for (const char *pointer = line.c_str(); *pointer != '\0'; ++pointer) {
                char symbol = *pointer;
```

```
            if (node->children.find(symbol) == node->children.end()) {
                node->children[symbol] = new TrieNode();
            }

            node = node->children[symbol];
        }

        node->index = next_index++;
    }

    int GetIndex(const std::string& line) {
        TrieNode* node = root;

        for (const char *pointer = line.c_str(); *pointer != '\0'; ++pointer) {
            char symbol = *pointer;

            if (node->children.find(symbol) == node->children.end()) {
                return -1;
            }

            node = node->children[symbol];
        }

        return node->index;
    }

    std::string GetStringByIndex(int index) {
        return GetString(root, "", index);
    }


private:
    TrieNode* root;
    int next_index;

    std::string GetString(TrieNode* node, const std::string& current, int index_number) {
        if (node->index == index_number) {
            return current;
        }
```

```
            std::string result;

            for (const auto& value : node->children) {
                result = GetString(value.second, current + value.first, index_number);

                if (result != "*") {
                    return result;
                }
            }

            return "*";
        }

};


std::vector<int> Compress(const std::string& text) {
    Trie trie;

    for (char symbol = 'a'; symbol <= 'z'; symbol++) {
        std::string line = std::string(1, symbol);
        trie.Insert(line);
    }

    std::string ending = "#";
    trie.Insert(ending);
    std::string current;
    std::vector<int> result;

    for (const char *pointer = text.c_str(); *pointer != '\0'; ++pointer) {
        char symbol = *pointer;
        std::string concatenation = current + symbol;
        int index = trie.GetIndex(concatenation);

        if (index != -1) {
            current = concatenation;
        } else {
            result.push_back(trie.GetIndex(current));
            trie.Insert(concatenation);
            current = symbol;
        }
```

```
    }

    if (!current.empty()) {
        result.push_back(trie.GetIndex(current));
    }

    return result;
}

std::string Decompress(const std::vector<int>& codes) {
    Trie trie;
    std::string result;
    std::string current;
    std::string symbol;

    for (char symbol = 'a'; symbol <= 'z'; symbol++) {
        std::string line = std::string(1, symbol);
        trie.Insert(line);
    }

    std::string ending = "#";
    trie.Insert(ending);

    for (int size = 0; size < codes.size(); size++) {
        std::string check = trie.GetStringByIndex(codes[size]);

        if (check != "*") {
            symbol = check;
        } else {
            symbol = current + current[0];
        }

        std::string concatenation = current + symbol[0];
        int index = trie.GetIndex(concatenation);

        if (index != -1) {
            current = concatenation;
        } else {
            result += current;
            trie.Insert(concatenation);
            current = symbol;
```

```
        }
    }

    result += current;

    return result;
}


int main() {
    std::string command, input;
    std::cin >> command;

    if (command == "compress") {
        std::cin >> input;
        std::vector<int> compressed = Compress(input + "#");

        for (size_t size = 0; size < compressed.size(); ++size) {
            int code = compressed[size];
            std::cout << code << " ";
        }
    } else if (command == "decompress") {
        std::vector<int> codes;
        int code;

        while (std::cin >> code) {
            codes.push_back(code);
        }

        std::string decompressed = Decompress(codes);

        if (decompressed[decompressed.size() - 1] == '#') {
            decompressed.pop_back();
        }

        std::cout << decompressed;
    }

    return 0;
}
```

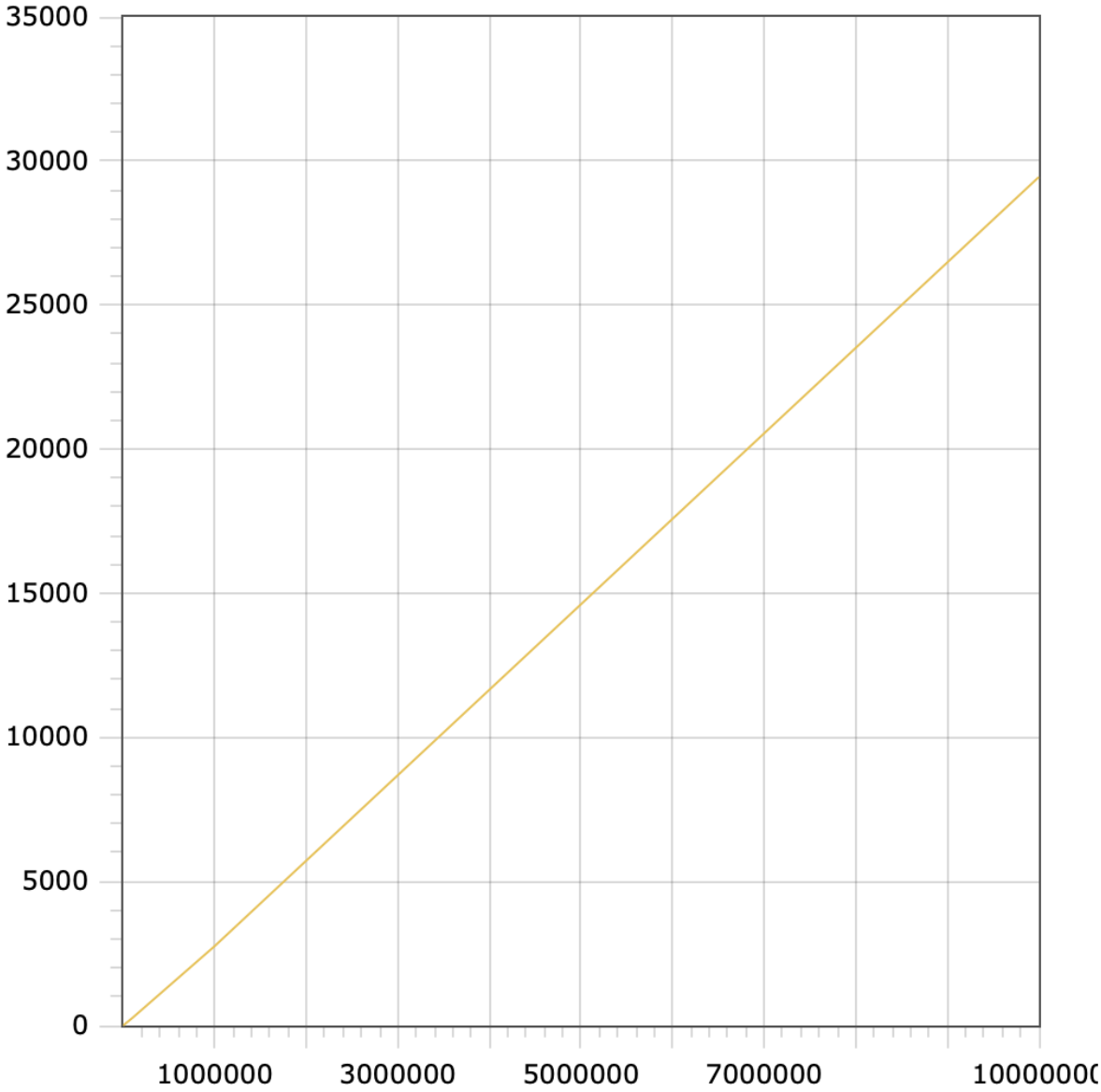
4. Демонстрация работы программы

```
MacBook-Pro-MacBook:~ macbookpro$ ./da_kp1
compress
abracadabra
0 1 17 0 2 0 3 27 29 26
MacBook-Pro-MacBook:~ macbookpro$ ./da_kp1
decompress
0 1 17 0 2 0 3 27 29 26
abracadabra
MacBook-Pro-MacBook:~ macbookpro$ ./da_kp1
compress
aaaaa
0 27 27 26
MacBook-Pro-MacBook:~ macbookpro$ ./da_kp1
decompress
0 27 27 26
aaaaa
MacBook-Pro-MacBook:~ macbookpro$
```

5. Тест производительности

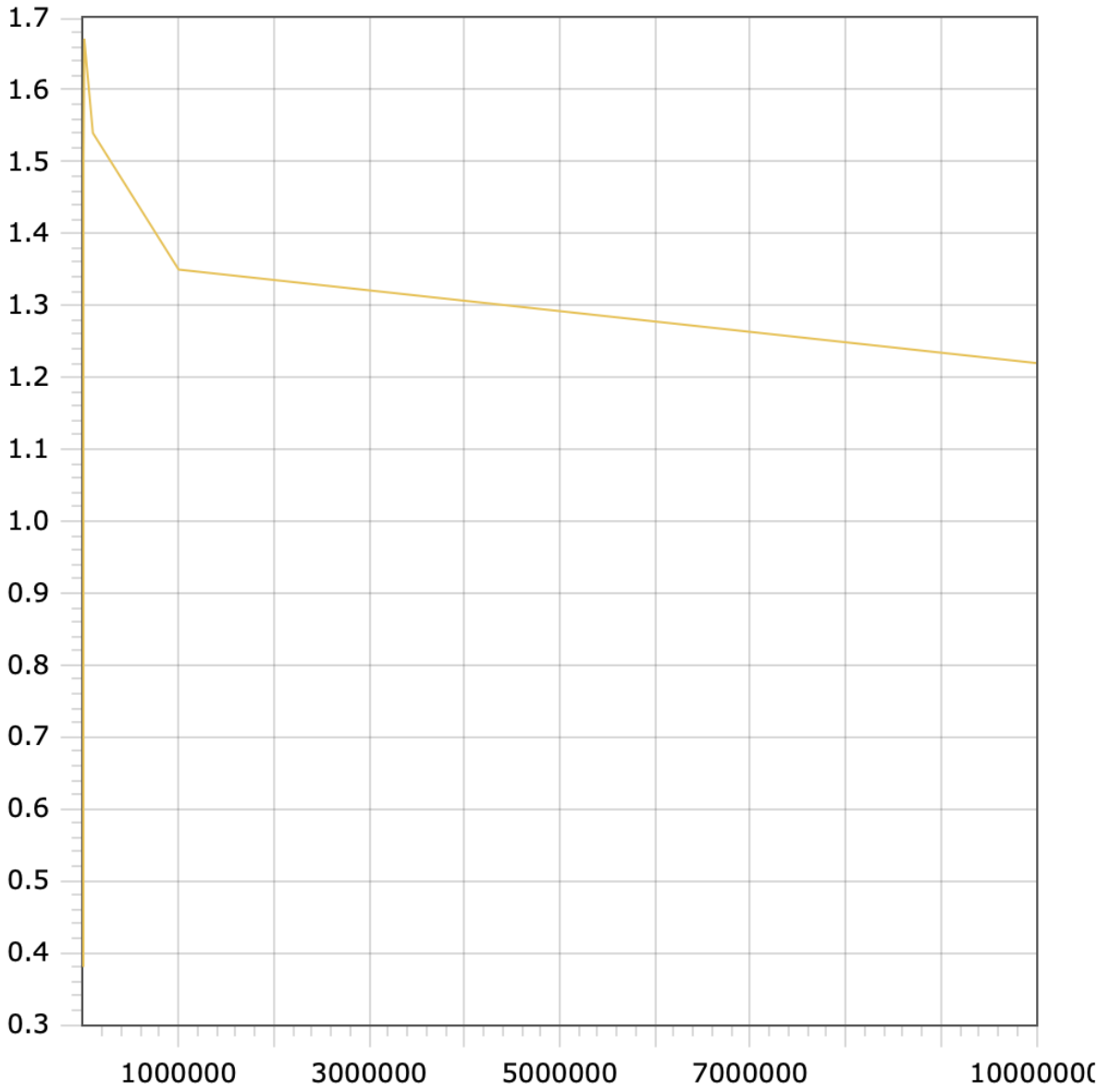
Проверим время работы архиватора в зависимости от длины входного текста:

Длина текста (количество символов)	Время работы программы (мс)
100	1
1000	3
10000	32
100000	264
1000000	2761
10000000	29444



Посмотрим на коэффициент сжатия при тексте различной длины (для каждого теста создается рандомная строка из маленьких латинских букв, которая повторяется 100 раз):

Длина текста (количество символов)	Размер начального файла (Кб)	Размер конечного файла (Кб)	Коэффициент сжатия
100	0,11	0,29	0,38
1000	1	0,46	1,5
10000	10	6	1,67
100000	100	65	1,54
1000000	1000	739	1,35
10000000	10000	8200	1,22



6. Вывод

В результате выполнения лабораторной работы я познакомилась с алгоритмом сжатия LZW, с помощью него реализовала кодирование и декодирование текста, проанализировала время работы программы в зависимости от длины входного текста.

7. Список литературы

1) Алгоритм Лемпеля – Зива - Велча

   URL: https://ru.wikipedia.org/wiki/Алгоритм_Лемпеля_—_Зива_—_Велча#Декодирование

   (дата последнего обновления 29.05.2023)

2) Алгоритм LZW

   URL: https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_LZW

   (дата последнего обновления 04.09.2022)