**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №3**

**по дисциплине «Построение и анализ алгоритмов»**

Тема: **Редакционное расстояние**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3341 |  | Ягудин Д.Р. |
| Преподаватель |  | Фирсов М.А. |

Санкт-Петербург

2025

## Цель работы

Целью данной лабораторной работы является закрепление навыков построения алгоритма Вагнера-Фишера для вычисления редакционного расстояния между двумя строками. Также необходимо реализовать предписание оптимальной последовательности правок (вставки, удаления, замены) для преобразования одной строки в другую.

**Задание**

Вариант 13:

Над строкой *ε* (будем считать строкой непрерывную последовательность из латинских букв) заданы следующие операции:

1. *replace(ε, a, b)* – заменить символ *a* на символ *b*.
2. *insert(ε, a)* – вставить в строку символ *a* (на любую позицию).
3. *delete(ε, b)* – удалить из строки символ *b*.

Каждая операция может иметь некоторую цену выполнения (положительное число).

Даны две строки *A* и *B*, а также три числа, отвечающие за цену каждой операции.

**Входные данные**: первая строка – три числа: цена операции replace, цена операции insert, цена операции delete; вторая строка – *A*; третья строка – *B*.

**Sample Input:**

1 1 1

entrance

reenterable

Задание 4.1. Определите минимальную стоимость операций, которые необходимы для превращения строки *A* в строку *B*.

**Выходные данные**: одно число – минимальная стоимость операций.

**Sample Output:**

5

Задание 4.2. Определите последовательность операций (редакционное предписание) с минимальной стоимостью, которые необходимы для превращения строки *A* в строку *B*.

| Пример (все операции стоят одинаково) | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **M** | **M** | **M** | **R** | **I** | **M** | **R** | **R** |
| **C** | **O** | **N** | **N** |  | **E** | **C** | **T** |
| C | O | N | **E** | **H** | E | A | D |

| Пример (цена замены 3, остальные операции по 1) | | | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **M** | **M** | **M** | **D** | **M** | **I** | **I** | **I** | **I** | **D** | **D** |
| **C** | **O** | **N** | **N** | **E** |  |  |  |  | **C** | **T** |
| C | O | N |  | E | **H** | **E** | **A** | **D** |  |  |

**Выходные данные**: первая строка – последовательность операций (M – совпадение, ничего делать не надо; R – заменить символ на другой; I – вставить символ на текущую позицию; D – удалить символ из строки); вторая строка – исходная строка A; третья строка – исходная строка B.

**Sample Output:**

IMIMMIMMRRM

entrance

reenterable

Индивидуализация:

Вывести не одно, а все редакционные предписания с минимальной стоимостью. Для одного из предписаний продемонстрировать его применение для преобразования 1-ой строки во 2-ую.

**Основные теоретические положения**

Расстояние Левенштейна (редакционное расстояние) – метрика, измеряющая по модулю разность между двумя последовательностями символов. Редакционное расстояние определяется как минимальное количество односимвольных операций (вставка, удаление, замена), необходимых для преобразования одной последовательности символов в другую. Операциям, используемым в этом преобразовании, можно назначить разную стоимость.

Редакционное предписание — это последовательность операций (вставка(I), удаление(D), замена(R), совпадение(M)), необходимых для минимального преобразования одной строки в другую.

**Описание алгоритма**

1. Определение минимальной стоимости.

Создается таблица *dp* размером *(n + 1) x (m + 1),* где *n* — длина строки *A*, *m* — длина строки *B*. Дополнительная строка и столбец используются для учета пустых строк (+ 1 к длинам строк при инициализации таблицы).

Для первой строки *dp*[0][j] и первого столбца *dp*[i][0] устанавливаются начальные значения:

*dp[0][j] = j \* inst* — если строка A пустая, для каждого символа строки B выполняется только операция вставки.

*table*[i][0] = *i \*del* — если строка B пустая, для каждого символа строки A выполняется только операция удаления.

После заполнения первой строки и первого столбца начинаем заполнять таблицу с *i = 1, j = 1.* Для каждой ячейки dp*[i][j]* вычисляем стоимость из трех возможных операций:

1. Вставка – стоимость вставки нового символа из строки B в строку A равна dp*[i][j – 1] +ins*.

2. Удаление – стоимость удаления символа из строки *A dp[i – 1][j] + del*.

3. Замена – Если символы строк не совпадают, то стоимость будет равна dp*[i - 1][j - 1]* + rep.

Если символы строк *A[i-1]* и *B[j-1*] равны, то стоимость этой операции равна dp*[i-1][j-1]* (т.е. не требует затрат).

В ячейку dp[i][j] записываем минимальное значение из этих операций (вставка, удаление, замена)

Минимальное редакционное расстояние между строками A и B будет содержаться в dp*[n][m].*

Асимптотика:

Время: *O*(*n \* m*) – где n - длина строки A, m – длина строки B.

Память: *O*(*n \* m*) – где n - длина строки A, m – длина строки B.

1. Определение редакционного предписания.

Создается строка *sequence*, которая будет хранить последовательность операций для преобразования строки *A* в строку *B*. Алгоритм начинается с правого нижнего угла таблицы dp*[n][m]*, который содержит минимальное редакционное расстояние между строками *A* и *B*.

Для каждой ячейки dp*[i][j]* вычисляем стоимость возможных операций, аналогично как при определении минимальной стоимости.

Запускаем цикл, пока один из индексов *i* или *j* не станет равным нулю, изначально i и j равны размеру строк *A* и *B*. В цикле проверяем значение каждой ячейки, если её значение совпадает со стоимостью вставки, то sequence добавляем символ “*I*” и сдвигаем указатель по строке *B*, если совпадает со стоимостью совпадений, то sequence добавляем символ “*M*” и сдвигаем указатели по строкам *A* и *B*, если совпадает со стоимостью замены, то добавляем к sequence “*R*” и сдвигаем указатели по строкам *A* и *B*, если совпадает со стоимостью удаления, то sequence добавляем “*D*” и сдвигаем по строке *A*.

Если хотим получить последовательность операций, то надо получить обратный *sequence*, так как мы шли с конца в начало.

Асимптотика:

Время: *O*(*n + m*) – где n - длина строки A, m – длина строки B.

Память: *O*(*n \* m*) – где n - длина строки A, m – длина строки B.

**Описание функций и структур данных**

1. std::string get\_moves(const std::vector<std::vector<Cell>>& matrix, int n, int m) – Функция нахождения редакционного предписания. На вход получает двумерный вектор, и размерность n и m. Возвращает строку, содержащую редакционное предписание.
2. std::set<std::string> get\_all\_moves(const std::vector<std::vector<Cell>>& matrix, int n, int m) – Функция нахождения всех редакционных предписаний, по длине равных самому оптимальному. На вход получает двумерный вектор, и размерность n и m. Возвращает множество строк редакционных предписаний.
3. void demonstrate\_transformation(const std::string& str1, const std::string& str2, const std::string& sequence) – Функция, демонстрирующая редактирование слова при помощи редакционного предписания. На вход принимает строки str1 и str2, а так же само редакционное предписание.
4. std::vector<std::vector<Cell>> Levenshtain(std::string str\_1, std::string str\_2, std::vector<int> flags) – Функция алгоритма Левенштейна. На вход принимает строки str\_1 и str\_2 для которых применяется алгоритм, и вектор флагов для настройки вывода промежуточных данных и установки своих собственных цен для каждой операции. Возвращает двумерный вектор.

**Тестирование**

Тестовые сценарии: 1. Одинаковые строки. 2. Замена дешевле удаления и вставки. 3. Вставка дешевле замены. 4. Обычный случай. 5. Тест, предложенный заданием. 6. Приоритет удаления/вставки над заменой.

Результаты тестирования представлены в табл. 1 и 2.

Таблица 1 – Результаты тестирования определения минимальной стоимости операций Задание № 4.1

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
|  | abc  abc | 0 | Тест 1: Одинаковые строки |
|  | a  b  1 10 10 | 1 | Тест 2: Замена дешевле удаления и вставки |
|  | abc  abxbc  10 1 10 | 2 | Тест 3: Вставка дешевле замены |
|  | kitten  sitting | 3 | Тест 4: Обычный случай |
| 1. a | entrance  reenterable | 5 | Тест 5: Тест, предложенный заданием |
|  | abcd  abef  3 2 1 | 6 | Тест 6: Приоритет удаления/вставки над заменой |

Таблица 2 – Результаты определения последовательности операций Задание № 4.2

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
|  | abc  abc | MMM | Тест 1: Одинаковые строки |
|  | a  b  1 10 10 | R | Тест 2: Замена дешевле удаления и вставки |
|  | abc  abxbc  10 1 10 | MMIIMM | Тест 3: Вставка дешевле замены |
|  | kitten  sitting | RMMMRMI | Тест 4: Обычный случай |
|  | entrance  reenterable | IMIMMIMMRRM | Тест 5: Тест, предложенный заданием |
|  | abcd  abef  3 2 1 | MMDDII | Тест 6: Приоритет удаления/вставки над заменой |

## Выводы

В ходе выполнения работы была реализована и протестирована эффективная реализация алгоритма Вагнера-Фишера для поиска редакционного расстояния и редакционного предписания. Алгоритм был протестирован на различных наборах тестов, что позволило удостовериться в его корректности и надежности. Работа алгоритма была визуализировано выводом в консоль.

# Приложение А Исходный код программы

Название файла: main.cpp

#include "includes/lib.h"

int main() {

    std::cout << "Input strings\n";

    std::string str\_1, str\_2;

    std::cin >> str\_1 >> str\_2;

    std::string input;

    std::vector<int> flags(2, 0);

    std::cout << "Wanna see how program works?(y/n)\n";

    std::cin >> input;

    if (input == "y"){

        flags[0] = 1;

    }

    input = "";

    std::cout << "Wanna use your own costs of operation?(y/n)\n";

    std::cin >> input;

    if (input == "y"){

        flags[1] = 1;

    }

    std::vector<std::vector<Cell>> matrix = Levenshtain(str\_1, str\_2, flags);

    return 0;

}

Название файла: lib.cpp

#include "lib.h"

std::string get\_moves(const std::vector<std::vector<Cell>>& matrix, int n, int m){

    std::string sequence;

    int i = m, j = n;

    while (i > 0 || j > 0) {

        std::cout << "Index i: " << i << " j: " << j << '\n';

        std::cout << "Move: " << matrix[i][j].move << '\n';

        switch (matrix[i][j].move) {

            case 'M':

                sequence += 'M';

                i--; j--;

                break;

            case 'R':

                sequence += 'R';

                i--; j--;

                break;

            case 'I':

                sequence += 'I';

                j--;

                break;

            case 'D':

                sequence += 'D';

                i--;

                break;

        }

    }

    std::reverse(sequence.begin(), sequence.end());

    return sequence;

}

std::set<std::string> get\_all\_moves(const std::vector<std::vector<Cell>>& matrix, int n, int m){

    std::set<std::string> result;

    if (n == 0 && m == 0) {

        result.insert("");

        return result;

    }

    std::vector<std::pair<int, int>> moves;

    int min\_cost = matrix[n][m].cost;

    // Проверяем все возможные ходы

    if (n > 0 && m > 0 && matrix[n-1][m-1].cost + (matrix[n][m].move == 'R' ? 1 : 0) == min\_cost) {

        moves.emplace\_back(n-1, m-1);

    }

    if (m > 0 && matrix[n][m-1].cost + 1 == min\_cost) {

        moves.emplace\_back(n, m-1);

    }

    if (n > 0 && matrix[n-1][m].cost + 1 == min\_cost) {

        moves.emplace\_back(n-1, m);

    }

    // Рекурсивно собираем все последовательности

    for (const auto& move\_pair : moves) {

        char move = ' ';

        if (move\_pair.first == n-1 && move\_pair.second == m-1) {

            move = (matrix[n][m].move == 'M' ? 'M' : 'R');

        } else if (move\_pair.first == n && move\_pair.second == m-1) {

            move = 'I';

        } else if (move\_pair.first == n-1 && move\_pair.second == m) {

            move = 'D';

        }

        auto sub\_moves = get\_all\_moves(matrix, move\_pair.first, move\_pair.second);

        for (const auto& sub\_move : sub\_moves) {

            result.insert(sub\_move + move);

        }

    }

    return result;

}

void printMatrix(const std::vector<std::vector<Cell>>& matrix){

    for (const auto& row : matrix) {

        for (Cell val : row) {

            std::cout << val.cost << "/" << val.move << " ";

        }

        std::cout << std::endl;

    }

}

void demonstrate\_transformation(const std::string& str1, const std::string& str2, const std::string& sequence){

    std::string current = str1;

    int pos1 = 0, pos2 = 0;

    std::cout << "Start:  " << current << "\n";

    for (size\_t i = 0; i < sequence.size(); ++i) {

        char move = sequence[i];

        std::string step = current;

        switch (move) {

            case 'M':

                step[pos1] = step[pos1] ;

                pos1++;

                pos2++;

                break;

            case 'R':

                step[pos1] = str2[pos2];

                pos1++;

                pos2++;

                break;

            case 'I':

                step.insert(pos1, 1, str2[pos2]);

                pos2++;

                break;

            case 'D':

                step.erase(pos1, 1);

                break;

        }

        std::cout << "Step " << i+1 << " (" << move << "): " << step << "\n";

        current = step;

    }

    std::cout << "Result: " << current << "\n";

}

std::vector<std::vector<Cell>> Levenshtain(std::string str\_1, std::string str\_2, std::vector<int> flags){

    int rep = 1, ins = 1, del = 1;

    if (flags[1] == 1){

        std::cout << "Input costs of operations(replace/insert/delete)\n";

        std::cin >> rep >> ins >> del;

    }

    int m = str\_1.size(), n = str\_2.size();

    std::vector<std::vector<Cell>> dp(m + 1, std::vector<Cell>(n + 1));

    for (int i = 0; i <= m; i++) {

        dp[i][0] = Cell(i \* del, 'D');  // Удаление всех символов из str\_1

    }

    for (int j = 0; j <= n; j++) {

        dp[0][j] = Cell(j \* ins, 'I');  // Вставка всех символов в str\_2

    }

    if (flags[0] == 1){

        std::cout << "Initialized matrix\n";

        printMatrix(dp);

    }

    // Заполнение матрицы

    for (int i = 1; i <= m; i++) {

        for (int j = 1; j <= n; j++) {

            if (str\_1[i - 1] == str\_2[j - 1]) {

                dp[i][j] = Cell(dp[i - 1][j - 1].cost, 'M');

                if (flags[0] == 1){

                    std::cout << "compared chars: " << str\_1[i - 1] << " " << str\_2[j - 1] << " " << dp[i][j].move << '\n';

                }

            } else {

                Cell replace\_cost = Cell(dp[i - 1][j - 1].cost + rep, 'R');

                Cell insert\_cost = Cell(dp[i][j - 1].cost + ins, 'I');

                Cell delete\_cost = Cell(dp[i - 1][j].cost + del, 'D');

                if (replace\_cost.cost <= insert\_cost.cost && replace\_cost.cost <= delete\_cost.cost) {

                    dp[i][j] = replace\_cost;

                } else if (insert\_cost.cost <= delete\_cost.cost) {

                    dp[i][j] = insert\_cost;

                } else {

                    dp[i][j] = delete\_cost;

                }

                if (flags[0] == 1){

                    std::cout << "compared chars: " << str\_1[i - 1] << " " << str\_2[j - 1] << " " << dp[i][j].move << '\n';

                }

            }

        }

    }

    if (flags[0] == 1){

        std::cout << "Result matrix:\n";

        // Если нужно посмотреть всю матрицу:

        printMatrix(dp);

    }

    // Вывод результата (минимальное расстояние)

    std::cout << dp[m][n].cost << std::endl;

    if (flags[0] == 1){

        std::cout << "All moves: ";

        // Вывод последоввательности действий

        std::cout << get\_moves(dp, n, m) << '\n';

    }

    // Вывод всех последовательностей

    auto all\_sequences = get\_all\_moves(dp, m, n);

    std::cout << "\nAll optimal sequences (" << all\_sequences.size() << "):\n";

    for (const auto& seq : all\_sequences) {

        std::cout << seq << "\n";

    }

    // Демонстрация преобразования

    if (!all\_sequences.empty()) {

        std::cout << "\nDemonstrating transformation for sequence: " << \*all\_sequences.begin() << "\n";

        demonstrate\_transformation(str\_1, str\_2, \*all\_sequences.begin());

    }

    return dp;

}

Название файла: lib.h

#ifndef LEVENSTEIN\_H

#define LEVENSTEIN\_H

#include <vector>

#include <string>

void getOperationSequence(

std::vector<std::vector<int>> table, // Таблица

std::vector<std::pair<int, int>>& path, // Вектор для хранения пути

std::vector<int> cost, // Стоимости операций: cost[0] - замена, cost[1] - вставка, cost[2] - удаление

std::string& sequence, // Строка, в которую записывается последовательность операций

std::string a, // Исходная строка

std::string b // Целевая строка

);

void levenstein(

std::vector<std::vector<int>>& table,

std::string a, // Исходная строка

std::string b, // Целевая строка

std::vector<int> cost = {1,1,1} // Стоимости операций: cost[0] - замена, cost[1] - вставка, cost[2] - удаление, по умолчанию все 1

);

#endif

Название файла: debug.cpp

#include <iostream>

#include <vector>

#include <string>

#include <algorithm>

#include <set>

class Cell{

    public:

    int cost;

    char move;

    Cell() : cost(0), move('\0') {}

    Cell(int new\_cost, char new\_move): cost(new\_cost), move(new\_move) {}

};

std::set<std::string> get\_all\_moves(const std::vector<std::vector<Cell>>& matrix, int n, int m);

std::string get\_moves(const std::vector<std::vector<Cell>>& matrix, int n, int m);

void printMatrix(const std::vector<std::vector<Cell>>& matrix);

void demonstrate\_transformation(const std::string& str1, const std::string& str2, const std::string& sequence);

std::vector<std::vector<Cell>> Levenshtain(std::string str\_1, std::string str\_2, std::vector<int> flags);