**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:

**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 *table* размером *(a + 1) x (b + 1),* где *a* — длина строки *A*, *b* — длина строки *B*. Дополнительная строка и столбец используются для учета пустых строк (+ 1 к длинам строк при инициализации таблицы).

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

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

*table*[i][0] = *i \* deleteCost* — 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 *table[i][j]* вычисляем стоимость из трех возможных операций:

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

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

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

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

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

Минимальное редакционное расстояние между строками A и B будет содержаться в *table[a][b].*

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 *table[a][b]*, который содержит минимальное редакционное расстояние между строками *A* и *B*.

Для каждой ячейки *table[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. Функция *void levenstein(std::vector<std::vector<int>>& table, std::string a, std::string b,std::vector<int> cost = {1,1,1})*

Назначение:

Реализация алгоритма редакционное расстояние Вагнера-Фишера. Вычисление редакционного расстояния между строками *a* и *b* с учетом заданных стоимостей операций вставки, удаления и замены. В результате таблица *table* заполняется минимальными стоимостями для каждой подстроки, и редакционное расстояние будет содержаться в правом нижнем углу таблицы.

Аргументы:

*std::vector<std::vector<int>>& table –* таблица для заполнения минимальными стоимостями преобразований подстрок строк *a* и *b*.

*std::string a* – первая строка для вычисления редакционного расстояния.

*std::string b* – вторая строка для вычисления редакционного расстояний.

*std::vector<int> cost={1, 1, 1}* – вектор содержащий стоимости операций: *cost[0]* – стоимость замены символа, *cost[1]* – стоимость вставки символа, *cost[2]* – стоимость удаления символа. По умолчанию все операции имеют стоимость 1.

2. Функция *void getOperationSequence*(*std::vector<std::vector<int>> table, std::vector<int> cost, std::string& sequence, std::string a,nstd::string b*)

Назначение:

Вычисление редакционного предписания – последовательность операций, необходимых для минимального преобразования строки *a* в строку *b* на основе предварительно вычисленной таблицы *table*.

Аргументы:

*std::vector<std::vector<int>> table –* таблица содержащая минимальные стоимости преобразования подстрок строки *a* в подстроки строки *b.*

*std::vector<std::pair<int, int>>& path* – вектор пар индексов *i* и *j* для хранения пути

*std::vector<int> cost*– вектор содержащий стоимости операций: *cost[0]* – стоимость замены символа, *cost[1]* – стоимость вставки символа, *cost[2]* – стоимость удаления символа.

**std::string& sequence** – строка, в которую записывается последовательность операций (“*M”* – совпадение, “I” – вставка, “D“– удаление, “R” – замена).

*std::string a* – первая строка, использованная в формировании таблицы.

*std::string b* – вторая строка, использованная в формировании таблицы.

3. Функция *void showTable(const std::vector<std::vector<int>>& table, const std::vector<std::pair<int, int>>& path, const std::string& a, const std::string& b*)

Назначение:

Отображение таблицы со стоимостями операций, в которой показываются ячейки, которые являются частью пути изменений и выделены эти ячейки в [].

Аргументы:

*const std::vector<std::vector<int>>& table –* таблица содержащая минимальные стоимости преобразования подстрок строки *a* в подстроки строки *b.*

*const std::vector<std::pair<int, int>>& path* – вектор содержит пары индексов, представляющие ячейки, которые участвуют в редакционном расстоянии(пути преобразований).

*const std::string& a* – исходная строка, которая будет преобразована в строку *b*.

*const std::string& b* – строка, в которую необходимо преобразовать исходную.

**Тестирование**

Для тестирования была реализована функция *void runTests*().

Функция runTests() предназначена для комплексной проверки алгоритма редакционных расстояний (Вагнера-Фишера) на различных сценариях преобразования строк. Каждый тестовый случай включает:

1. Стоимости операций замены (replace), вставки (insert) и удаления (delete).

2. Исходную строку A и целевую строку B.

3. Ожидаемую минимальную стоимость преобразования.

4. Ожидаемую последовательность операций (в виде строки из символов R, I, D, M).

Тестовые сценарии: 1. Одинаковые строки. 2. Пустые строки. 3. Пустая строка преобразование в непустую. 4. Непустая строка преобразование в пустую. 5. Замена дешевле удаления и вставки. 6. Вставка дешевле замены. 7. Обычный случай. 8. Тест, предложенный заданием. 9. 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 |
|  | 1 1 1  abc  abc | 0 | Тест 1: Одинаковые строки |
|  | 5 3 2 | 0 | Тест 2: Пустые строки |
|  | 2 1 3  xyz | 3 | Тест 3: Пустая строка преобразование в непустую |
|  | 4 5 2  a b c | 6 | Тест 4: Непустая строка преобразование в пустую |
|  | 1 5 5  a  b | 1 | Тест 5: Замена дешевле удаления и вставки |
|  | 5 1 5  abc  abxbc | 2 | Тест 6: Вставка дешевле замены |
|  | 1 1 1  kitten  sitting | 3 | Тест 7: Обычный случай |
| 1. a | 1 1 1  entrance  reenterable | 5 | Тест 8: Тест, предложенный заданием |
|  | 3 2 1  abcd  abef | 6 | Тест 9: Приоритет удаления/вставки над заменой |

Таблица 2 – Результаты определения последовательности операций Задание № 4.2

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
|  | 1 1 1  abc  abc | MMM  abc  abc  1 | Тест 1: Одинаковые строки |
|  | 5 3 2 | 3 Пустые строки | Тест 2: Пустые строки |
|  | 2 1 3  xyz | III  xyz | Тест 3: Пустая строка преобразование в непустую |
|  | 4 5 2  a b c | DDD  abc | Тест 4: Непустая строка преобразование в пустую |
|  | 1 5 5  a  b | R  a  b | Тест 5: Замена дешевле удаления и вставки |
|  | 5 1 5  abc  abxbc | MMIIMM  abc  abxbc | Тест 6: Вставка дешевле замены |
|  | 1 1 1  kitten  sitting | RMMMRMI  kitten  sitting | Тест 7: Обычный случай |
|  | 1 1 1  entrance  reenterable | IMIMMIMMRRM  entrance  reenterable | Тест 8: Тест, предложенный заданием |
|  | 3 2 1  abcd  abef | MMDDII  abcd  abef | Тест 9: 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 <iostream>

#include <vector>

#include <string>

#include <algorithm>

#include <iomanip>

#include <windows.h>

#include <set>

#include "debug.h"

#include "levenstein.h"

#define TASK\_MINCOST false

int main() {

SetConsoleOutputCP(CP\_UTF8);

std::vector<int> cost(3,0);

std::cin >> cost[0] >> cost[1] >> cost[2];

std::string a, b, sequence;

std::cin >> a >> b;

std::vector<std::vector<int>> table(a.size() + 1, std::vector<int>(b.size() + 1,0));

std::set<std::pair<int, int>> path;

levenstein(table, a, b, cost);

getOperationSequence(table, path, cost, sequence, a, b);

#if(DEBUG == true)

printOperations(table, path, sequence, a, b);

#endif

#if(TASK\_MINCOST == true)

std::cout << table.back().back() << "\n";

#else

std::cout << sequence << "\n";

std::cout << a << "\n";

std::cout << b << "\n";

runTests();

#endif

return 0;

}

Название файла: levenstein.cpp

#include "levenstein.h"

#include <algorithm>

#include <iostream>

#include "debug.h"

void getOperationSequence(

std::vector<std::vector<int>> table, // Двумерная таблица расстояний

std::vector<std::pair<int, int>>& path, // Порядок операций

std::vector<int> cost, // Стоимость операций (вставка, удаление, замена)

std::string& sequence, // Строка с последовательностью операций

std::string a, // Исходная строка

std::string b // Целевая строка

)

{

// Индексы текущих позиций в строках

int i = static\_cast<int>(a.size());

int j = static\_cast<int>(b.size());

#if (DEBUG == true)

// Строка для отслеживания текущего состояния

std::string current = a;

int offset = 0; // Смещение для учета изменений в текущей строке (добавление, удаление)

int step = 0; // Шаг выполнения

std::string outputSequence; // Строка для накопления последовательности операций

// Вывод начальной информации

std::cout << "Transforming \"" << a << "\" to \"" << b << "\":\n";

std::cout << "Current string: " << current << "\n";

std::cout << "Distance table (initial state):\n";

showTable(table, {}, a, b);

std::cout << "─────────────────────────────────────────\n";

#endif

// Алгоритм поиска редакционного предписания

while (i > 0 || j > 0) {

// Вычисление стоимости различных операций

int matchCost = (i > 0 && j > 0) ? table[i - 1][j - 1] : INT\_MAX;

int insertCost = (j > 0) ? table[i][j - 1] + cost[1] : INT\_MAX;

int deleteCost = (i > 0) ? table[i - 1][j] + cost[2] : INT\_MAX;

int replaceCost = (i > 0 && j > 0) ? table[i - 1][j - 1] + cost[0] : INT\_MAX;

// Инициализация переменной для операции

char op;

int iOld = i, jOld = j; // Сохраняем старые индексы для отладки

// Определение операции на основе стоимости

if (j > 0 && table[i][j] == insertCost) {

// Вставка

op = 'I';

j--; // двигаем указатель по строке b

} else if (i > 0 && j > 0 && a[i - 1] == b[j - 1] && table[i][j] == matchCost) {

// Совпадение

op = 'M';

i--; j--; // двигаем указатели по обеим строкам

} else if (i > 0 && j > 0 && table[i][j] == replaceCost) {

// Замена

op = 'R';

i--; j--; // Двигаем указатели по обеим строкам

} else if (i > 0 && table[i][j] == deleteCost) {

// Удаление

op = 'D';

i--; // Двигаем указатель по строке a

} else {

// Прерываем цикл, если нет подходящей операции

break;

}

// Добавляем операцию в итоговую последовательность

sequence += op;

path.emplace\_back(iOld, jOld); // Сохраняем старые индексы для пути

#if (DEBUG == true)

std::string explanation; // Описание операции

std::string newCurrent = current; // Обновлённая строка

std::string marker(current.size(), ' '); // Место для пометок изменений

int pos; // Позиция, где будет происходить изменение

switch (op) {

case 'I': {

pos = iOld;

char c = b[jOld - 1];

newCurrent.insert(pos, 1, c);

marker.insert(pos, 1, '^');

explanation = "Insert '" + std::string(1, c) + "' at pos " + std::to\_string(pos) + " (needed to get '" + b + "')";

offset++;

break;

}

case 'D': {

pos = iOld - offset;

char c = current[pos];

newCurrent.erase(pos, 1);

marker[pos] = '^';

explanation = "Remove extra character '" + std::string(1, c) + "' at pos " + std::to\_string(pos);

offset--;

break;

}

case 'R': {

pos = iOld - 1;

char from = current[pos]; // Символ для замены

char to = b[jOld - 1]; // Новый символ

newCurrent[pos] = to; // Заменяем символ

marker[pos] = '^';

explanation = "Replace '" + std::string(1, from) + "' with '" + std::string(1, to) + "' at pos " + std::to\_string(pos);

break;

}

case 'M': {

// Совпадение

pos = iOld - 1;

char c = current[pos]; // Совпавший символ

marker[pos] = '^';

explanation = "Character '" + std::string(1, c) + "' at pos " + std::to\_string(pos) + " is already matched";

break;

}

}

// Выводим информацию для отладки

outputSequence += op;

std::cout << "Current sequence operation " << outputSequence << "\n";

std::cout << "Step " << step + 1 << ": " << explanation << "\n";

std::cout << "Changes:\n";

std::cout << " Transforming to: \"" << b << "\"\n";

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

std::cout << " After: \"" << newCurrent << "\"\n";

std::cout << " " << marker << " (changed position)\n\n";

std::cout << "Current table state:\n";

std::cout << "• Cells in [] - already processed steps\n";

std::cout << "• Current cell: [" << iOld << "," << jOld << "]\n";

// Печатаем текущее состояние таблицы

std::vector<std::pair<int, int>> currentPath(path.begin(), path.end());

showTable(table, currentPath, a, b);

std::cout << "─────────────────────────────────────────\n";

// Обновляем текущую строку

current = newCurrent;

step++;

#endif

}

// Разворачиваем последовательность операций, так как мы шли от конца к началу

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

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

#if (DEBUG == true)

// Печатаем финальный результат

std::cout << "Final result: \"" << current << "\"\n";

std::cout << "Target was: \"" << b << "\"\n";

std::cout << (current == b ? "✓ Transformation successful!" : "⚠ Differences remain") << "\n";

#endif

}

void levenstein(

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

std::string a, // Исходная строка

std::string b, // Целевая строка

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

)

{

if (a.empty() && b.empty()) {

table[0][0] = 0;

return;

}

// Заполнение первой строки операцией вставки

for (int i = 1;i < table[0].size();i++) {

table[0][i] = table[0][i - 1] + cost[1];

#if(DEBUG == true)

std::cout << "Step [" << 0 << ", " << i << "]:\n";

std::cout << "Insert selected: '" << b[i - 1] << "' into target string\n";

showTable(table, {{0, i}}, a ,b);

std::cout << "─────────────────────────────────────────\n";

#endif

}

// Заполнение первого столбца операцией удаления

for (int j = 1;j < table.size();j++) {

table[j][0] = table[j - 1][0] + cost[2];

#if(DEBUG == true)

std::cout << "Step [" << j << ", " << 0 << "]:\n";

std::cout << "Delete selected: '" << a[j - 1] << "' into source string\n";

showTable(table, {{j,0}}, a ,b);

std::cout << "─────────────────────────────────────────\n";

#endif

}

// Заполнение таблицы для всех остальных элементов начиная с 1,1

for (int i = 1;i < table.size();i++) {

for (int j = 1;j < table[i].size();j++) {

table[i][j] = std::min({table[i][j - 1] + cost[1], // вставка

table[i - 1][j] + cost[2], // удаление

table[i-1][j-1] + ((a[i - 1] == b[j - 1]) ? 0 :cost[0])}); // замена(при совпадении 0)

#if(DEBUG == true)

std::cout << "Step [" << i << ", " << j << "]:\n";

if (table[i][j] == table[i - 1][j - 1] + cost[0]) std::cout << "Replace selected: '" << a[i - 1] << "'" << " with '" << b[j -1] << "'\n";

else if (table[i][j] == table[i][j - 1] + cost[1]) std::cout << "Insert selected: '" << b[j - 1] << "' into target string\n";

else if (table[i][j] == table[i - 1][j] + cost[2]) std::cout << "Delete selected: '" << a[i - 1] << " from source string\n";

else std::cout << "Match selected: '" << a[i - 1] << "' == '" << b[j -1] << "'\n";

std::cout << " • Insert cost " << table[i][j - 1] + cost[1] << "\n";

std::cout << " • Delete cost " << table[i - 1][j] + cost[2] << "\n";

std::cout << " • Replace cost " << table[i - 1][j - 1] + cost[0] << "\n";

std::cout << " • Match cost " << table[i - 1][j - 1] << "\n";

std::cout << "Fill cell: [" << i << ", " << j << "] by cost " << table[i][j] << "\n";

showTable(table, {{i,j}}, a ,b);

std::cout << "─────────────────────────────────────────\n";

#endif

}

}

}

Название файла: levenstein.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 "debug.h"

#include <iostream>

#include <iomanip>

#include <tuple>

#include <levenstein.h>

#include <cassert>

#include <algorithm>

void showTable(

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

const std::vector<std::pair<int, int>>& path,

const std::string& a,

const std::string& b)

{

// Находим максимальное число в таблице для определения ширины столбцов

int maxElem = 0;

for (const auto& row : table) {

for (int val : row) {

if (val > maxElem) {

maxElem = val;

}

}

}

int width = std::to\_string(maxElem).length() + 4; // +4 для учета скобок []

// Выводим заголовок (верхнюю строку)

std::cout << " " << std::setw(width);

std::cout << " " << std::setw(width);

for (char c : b) {

std::cout << c << std::setw(width);

}

std::cout << "\n";

// Выводим саму таблицу

for (int i = 0; i < table.size(); ++i) {

// Выводим символ из a или ' ' для первой строки

if (i == 0) {

std::cout << " " << std::setw(width);

}

else {

std::cout << a[i - 1] << std::setw(width);

}

// Выводим значения ячеек

for (int j = 0; j < table[i].size(); ++j) {

if (std::find(path.begin(), path.end(), std::make\_pair(i, j)) != path.end()) {

std::cout << "[" << table[i][j] << "]" << std::setw(width - 2);

}

else {

std::cout << table[i][j] << std::setw(width);

}

}

std::cout << "\n";

}

}

void runTests() {

std::vector<std::tuple<std::vector<int>, std::string, std::string, int, std::string>> tests = {

{{1, 1, 1}, "entrance", "reenterable", 5,"IMIMMIMMRRM"},

{{1,1,1}, "abc", "abc", 0, "MMM"},

{{5,3, 2}, "", "", 0, ""},

{{2, 1, 3}, "", "xyz", 3, "III"},

{{4, 5, 2}, "abc", "", 6, "DDD"},

{{1, 5, 5}, "a", "b", 1, "R"},

{{5, 1, 5}, "abc", "abxbc", 2, "MMIIM"},

{{1, 1, 1}, "kitten", "sitting", 3, "RMMMRMI"},

{{3, 2, 1}, "abcd", "abef", 6, "MMDDII"}

};

for (const auto& [cost, a, b, expected, sequence] : tests) {

std::vector<std::vector<int>> table(a.size() + 1, std::vector<int>(b.size() + 1, 0));

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

std::string seq;

levenstein(table, a, b, cost);

getOperationSequence(table, path, cost, seq, a, b);

std::cout << "Test:\n"

<< " A: \"" << a << "\"\n"

<< " B: \"" << b << "\"\n"

<< " Expected cost: " << expected << "\n"

<< " Calculated cost: " << table.back().back() << "\n"

<< " Expected Sequence: " << sequence << "\n"

<< " Sequence: " << seq << "\n";

assert(table.back().back() == expected);

assert(seq == sequence);

std::cout << " [✓]\n\n";

}

std::cout << "All test passed\n";

}

Название файла: debug.h

#ifndef DEBUG\_H

#define DEBUG\_H

#include <vector>

#include <string>

#define DEBUG true

void showTable(

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

const std::vector<std::pair<int, int>>& path,

const std::string& a,

const std::string& b

);

void runTests();

#endif