**ОТЧЕТ**

По лабораторной работе №1

По курсу «Анализ алгоритмов»

Тема: «Расстояние Левенштейна»

|  |  |
| --- | --- |
|  | **Министерство образования и науки Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

Студент: Зейналов З. Г.

Группа: ИУ7-51

Преподаватель: Волкова Л.Л.

Москва, 2019г.

Оглавление

[Введение 3](#_Toc20931656)

[Задачи работы: 4](#_Toc20931657)

[1.1. Описание алгоритмов 5](#_Toc20931658)

[2. Конструкторская часть 6](#_Toc20931659)

[2.1. Разработка алгоритмов 6](#_Toc20931660)

[3. Технологическая часть 7](#_Toc20931661)

[3.1. Требования к программному обеспечению 7](#_Toc20931662)

[3.2. Средства реализации 7](#_Toc20931663)

[3.3. Листинг кода 8](#_Toc20931664)

[3.4. Описание тестирования 8](#_Toc20931665)

[4. Экспериментальная часть 8](#_Toc20931666)

[4.1. Примеры работы 11](#_Toc20931667)

[4.2. Результаты тестирования 13](#_Toc20931668)

[4.3. Постановка эксперимента по замеру времени [и памяти] 14](#_Toc20931669)

[4.4. Сравнительный анализ на материале экспериментальных данных 14](#_Toc20931670)

[Заключение 15](#_Toc20931671)

# Введение

**Расстояние Левинштейна** - минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую. Измеряется для двух строк, широко используется в теории информации и компьютерной лингвистике.

Впервые задачу поставил в 1965 году советский математик Владимир Левенштейн при изучении последовательностей, впоследствии более общую задачу для произвольного алфавита связали с его именем. Большой вклад в изучение вопроса внёс Дэн Гасфилд.

**Расстояние Дамерау-Левенштейна**  - Эта вариация вносит в определение расстояния Левенштейна еще одно правило — транспозиция (перестановка) двух соседних букв также учитывается как одна операция, наряду со вставками, удалениями и заменами.

Еще пару лет назад Фредерик Дамерау мог бы гарантировать, что большинство ошибок при наборе текста — как раз и есть транспозиции. Поэтому именно данная метрика дает наилучшие результаты на практике.

# Задачи работы:

1) изучение алгоритмов Левенштейна и Дамерау-Левенштейна нахождения расстояния между

строками;

2) применение метода динамического программирования для матричной реализации указанных

алгоритмов;

3) получение практических навыков реализации указанных алгоритмов: двух алгоритмов в

матричной версии и одного из алгоритмов в рекурсивной версии;

4) сравнительный анализ линейной и рекурсивной реализаций выбранного алгоритма

определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);

5) экспериментальное подтверждение различий во временнóй эффективности рекурсивной и

нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при

помощи разработанного программного обеспечения на материале замеров процессорного времени

выполнения реализации на варьирующихся длинах строк;

6) описание и обоснование полученных результатов в отчете о выполненной лабораторной

работе, выполненного как расчётно-пояснительная записка к работе.

1. **Аналитическая часть**

В данном разделе будут предтавлены описания алгоритмов, формулы и оценки сложностей алгоритмов

## Описание алгоритмов

**Расстояние Левинштейна** или редакционное расстояние между строками s1 и s2 как min количество редакторских операций, необходимых для преобразования одной строки в другую.

Редакторские операции:

Формула для нахождения расстояния с использованием матрицы:

**Расстояние Дамерау-Левинштейна** вводится дополнительная операция перестановки или транспозиция, 2 буквы, стоимость = 1. Если индексы позволяют, и если соседние буквы совпадает с , то в минимум включается перестановка (транспозиция).

Применение алгоритмов:

* Поиск по словарю (частная задача лингвистики)
* Биоинформатика
* Поисковые системы, для предложения более подходящего запроса, в случае, если пользователь допустил ошибку или ввел одну букву раньше другой.

Данные алгоритмы так же допускают рекурсивную реализацию.

## Конструкторская часть

В данном разделе будут размещены схемы алгоритмов и сравнительный анализ рекурсивной и не рекурсивной реализаций.

## Разработка алгоритмов

**// сюда схемы алгоритмов**

* 1. **Сравнительный анализ рекурсивной и нерекурсивной реализаций**

Рекурсивный алгоритм, по сравнению с матричной реализацией работает медленнее, так как при исчерпывание всех возможных комбинаций, возникает ситуация, когда рекурсивные вызовы функций абсолютно идентичны.

Покажем это на примере:

## Технологическая часть

В данном разделе будут приведены Требования к программному обеспечению, средства реализации, листинг кода и примеры тестирования.

### Требования к программному обеспечению

На вход подаются 2 строки, на выходе необходимо получить матрицу и 3 результата, выдаваемых матричными реализациями обоих алгоритмов и рекурсивной реализации алгоритма Дамерау-Левенштейна.

Требуется замерить время работы каждой реализации.

### Средства реализации

В качестве языка программирования был выбран С++ в связи с его широким функционалом и быстротой работы, а так же благодаря привычному для меня синтаксису и семантики языка. Среда разработки - Qt. Время работы процессора замеряется с помощью функции:

unsigned long long **tick**(void)

{

unsigned long long d;

*\_\_asm\_\_* *\_\_volatile\_\_* ("rdtsc" : "=A" (d) );

*return* d;

}

Листинг 1 – Функция замера времени

### Листинг кода

void **fillBaseMatrix**(int s1, int s2, int \*\*matrix)

{

*for* (int i = 1; i < s1 + 1; i++)

{

matrix[i][0] = i;

}

*for* (int j = 1; j < s2 + 1; j++)

{

matrix[0][j] = j;

}

}

int **LevensteinMatrix**(string s1, string s2)

{

int row\_count = s1.length();

int column\_count = s2.length();

int \*\*matrix = allocateMatrix(row\_count, column\_count);

fillBaseMatrix(s1.length(), s2.length(), matrix);

int cost = 0;

*for* (int i = 1; i < row\_count + 1; i++)

{

*for* (int j = 1; j < column\_count + 1; j++)

{

*if* (s1[i] == s2[j])

{

cost = 0;

}

*else*

{

cost = 1;

}

matrix[i][j] = minimum(matrix[i - 1][j] + 1, matrix[i][j - 1] + 1, matrix[i - 1][j - 1] + cost);

}

}

cout << "=== Levenstein Matrix ===" << endl;

printMatrix(row\_count, column\_count, matrix);

int result = matrix[s1.length()][s2.length()];

clearMatrix(matrix, row\_count);

*return* result;

}

### Описание тестирования

**// описать, какие тесты будут проведены**

Листинг 2 – Реализация Левенштейна с помощью матрицы

int **DamerauLevensteinRecursion**(string s1, string s2)

{

*if* (s1.length() == 0)

*return* s2.length();

*if* (s2.length() == 0)

*return* s1.length();

int cost = 0;

*if* (s1[s1.length() - 1] == s2[s2.length() - 1])

cost = 0;

*else*

cost = 1;

int result = minimum(DamerauLevensteinRecursion(s1.substr(0, s1.length() - 1), s2) + 1, DamerauLevensteinRecursion(s1, s2.substr(0, s2.length() - 1)) + 1,

DamerauLevensteinRecursion(s1.substr(0, s1.length() - 1), s2.substr(0, s2.length() - 1)) + cost);

*if* (((s1.length() > 2 && s2.length() > 2) && s1[s1.length() - 1] == s2[s2.length() - 2]

&& s2[s2.length() - 1] == s1[s1.length() - 2]))

{

result = min(result, DamerauLevensteinRecursion(s1.substr(0, s1.length() - 2), s2.substr(0, s2.length() - 2)) + cost);

}

*return* result;

}

### Описание тестирования

**// описать, какие тесты будут проведены**

Листинг 3 – Реализация Дамерау-Левенштейна рекурсивно

int **DamerauLevinsteinMatrix**(string s1, string s2)

{

int row\_count = s1.length();

int column\_count = s2.length();

int \*\*matrix = allocateMatrix(row\_count, column\_count);

fillBaseMatrix(s1.length(), s2.length(), matrix);

int cost = 0;

*for* (int i = 1; i < row\_count + 1; i++)

{

*for* (int j = 1; j < column\_count + 1; j++)

{

*if* (s1[i] == s2[j])

{

cost = 0;

}

*else*

{

cost = 1;

}

matrix[i][j] = minimum(matrix[i - 1][j] + 1, matrix[i][j - 1] + 1, matrix[i - 1][j - 1] + cost);

*if* ((i > 1 && j > 1) && s1[i] == s2[j - 1] && s2[j] == s1[i - 1])

{

matrix[i][j] = min(matrix[i][j], matrix[i - 2][j - 2] + cost);

}

}

}

*//printMatrix(row\_count,* *column\_count,* *matrix);*

int result = matrix[s1.length()][s2.length()];

clearMatrix(matrix, row\_count);

*return* result;

}

Листинг 4 – Рализация алгоритма Дамерау-Левенштейна

с помощью матрицы

### Описание тестирования

Тестирование будет проведено на следующих примерах, включая пустые строки и строки, содержащие пробелы:

1) Развлечения, Увлечение

2) Скат, Кот

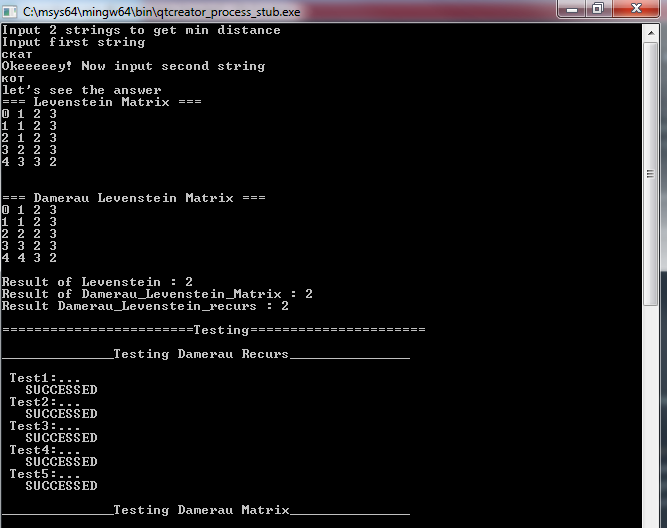
3) “ “, “ “

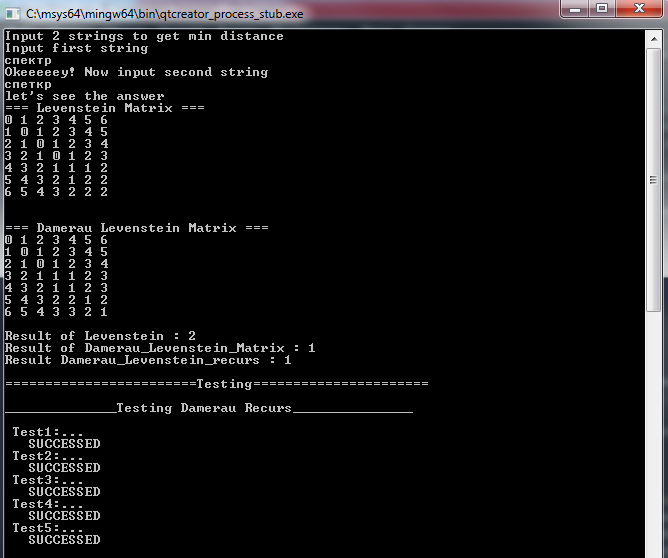
4) “ “, “ “  
5) “Darek”, “ “

## Экспериментальная часть

В данном разделе будут приведены примеры работы программы, постановка эксперимента и сравнительный анализ алгоритмов на основе экспериментальных данных.

### Примеры работы





### Результаты тестирования

int **mainTest**(void)

{

string strings[10] = {"Razvlecheniya", "Uvlechenie", "Skat", "kot",

" ", " ", "Darek", "", "", ""};

cout << "\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_Testing Damerau Recurs\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n" << endl;

DamerauLevensteinRTesting(strings);

cout << "\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_Testing Damerau Matrix\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n" << endl;

DamerauLevensteinMTesting(strings);

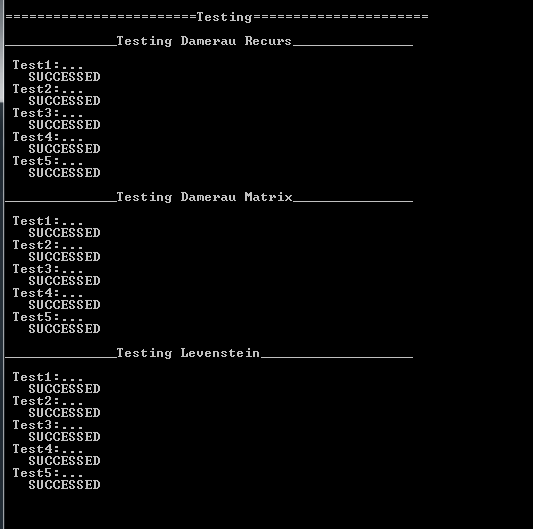
cout << "\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_Testing Levenstein\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n" << endl;

LevensteinMTesting(strings);

*return* 0;

}

Листинг 5 – Функция тестирования



### Постановка эксперимента по замеру времени [и памяти]

Для произведения замеров времени выполнения реализаций алгоритмов будет использована следующая формула , где t – время выполнения, N – количество замеров. Неоднократное измерение времени необходимо для построения более гладкого графика.

### Сравнительный анализ на материале экспериментальных данных

**// здесь эксперименты+выводы**

# Заключение

В ходе работы были изучены и реализованы в матричной форме алгоритмы Левенштейна и Дамерау-Левенштейна и рекурсивно алгоритм Дамерау-Левенштейна.