**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: Лумбунов Д.В.

Группа: ИУ7-54

Преподаватель: Волкова Л.Л.

Москва, 2019г.

Оглавление

[Введение 3](#_Toc21519577)

[Задачи работы: 4](#_Toc21519578)

[1. Аналитическая часть 5](#_Toc21519579)

[1.1 Описание алгоритмов 5](#_Toc21519580)

[2. Конструкторская часть 6](#_Toc21519581)

[2.1 Разработка алгоритмов 7](#_Toc21519582)

[2.2 Сравнительный анализ рекурсивной и нерекурсивной реализаций 13](#_Toc21519583)

[3 Технологическая часть 14](#_Toc21519584)

[3.1 Требования к программному обеспечению 14](#_Toc21519585)

[3.2 Средства реализации 14](#_Toc21519586)

[3.3 Листинг кода 15](#_Toc21519587)

[3.4 Описание тестирования 17](#_Toc21519588)

[4 Экспериментальная часть 18](#_Toc21519589)

[4.1 Примеры работы 18](#_Toc21519590)

[4.2 Результаты тестирования 20](#_Toc21519591)

[4.3 Постановка эксперимента по замеру времени [и памяти] 21](#_Toc21519592)

[4.4 Сравнительный анализ на материале экспериментальных данных 22](#_Toc21519593)

[Заключение 23](#_Toc21519594)

# 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 представлены описания алгоритмов, формулы и оценки сложностей алгоритмов

### Описание алгоритмов

**Рекурсивный алгоритм (Левенштейн)**

Здесь и далее считается, что элементы строк нумеруются с первого, как принято в математике, а не с нулевого, как принято во многих языках программирования.

Пусть S1{\displaystyle S\_{1}} и S2 {\displaystyle S\_{2}}SSЫ— две строки (длиной M и N MЬ соответственно) над некоторым алфавитом, тогда редакционное расстояние (расстояние Левенштейна) d(S1{\displaystyle S\_{1}}, S2) можно подсчитать по следующей рекуррентной формуле

d(S1{\displaystyle S\_{1}}, S2) = D(M,N), где

(1)

Рекурсивный алгоритм напрямую реализует эту формулу.

Функция (1) составлена из следующих соображений:

1. Для перевода из пустой строки в пустую требуется ноль операций.
2. Для перевода из пустой строки в строку 𝑎 требуется |𝑎| операций. Аналогично, для перевода из строки 𝑎 в пустую требуется |𝑎| операций.
3. Для перевода из строки 𝑎 в строку 𝑏 требуется выполнить последовательно некоторое кол-во операций (удаление, вставка, замена) в некоторой последовательности. Как можно показать сравнением, последовательность проведения любых двух операций можно поменять, и, как следствие, порядок проведения операций не имеет никакого значения. Тогда цена преобразования из строки 𝑎 в строку 𝑏 может быть выражена как (полагая, что 𝑎′, 𝑏′ — строки 𝑎 и 𝑏 без последнего символа соответственно):
   1. Сумма цены преобразования строки 𝑎′ в 𝑏 и цены проведения операции удаления, которая необходима для преобразования 𝑎 в 𝑎′;
   2. Сумма цены преобразования строки 𝑎 в 𝑏′ и цены проведения операции вставки, которая необходима для преобразования 𝑏′ в 𝑏;
   3. Сумма цены преобразования из 𝑎′ в 𝑏′ и операции замены, предполагая, что 𝑎 и 𝑏 оканчиваются разные символы;
   4. Цена преобразования из 𝑎′ в 𝑏′, предполагая, что 𝑎 и 𝑏 оканчиваются на один и тот же символ.

Очевидно, что минимальной ценой преобразования будет минимальное

значение этих вариантов.

**Алгоритм Вагнера-Фишера (построчный)**

Прямая реализация формулы (1) может быть малоэффективна при больших 𝑖, 𝑗, т. к. множество промежуточных значений 𝐷(𝑖, 𝑗), 1 ≤ 𝑖 ≤ |𝑎|, 1 ≤ 𝑗 ≤ |𝑏| вычисляются заново множество раз подряд. Для оптимизации нахождения расстояния Левенштейна предложено использовать матрицу (двумерный массив) в целях хранения соответствующих промежуточных значений. В таком случае алгоритм представляет собой построчное заполнение матрицы 𝐴|𝑎|,|𝑏| значениями 𝐷(𝑖, 𝑗) по формуле (1).

Можно заметить, что при каждом заполнении новой строки значения предыдущей становятся ненужными. Поэтому можно провести оптимизацию по памяти и использовать дополнительно только одномерный массив размером min(|𝑎|, |𝑏|). Такой вариант алгоритма называется построчным и именно он реализован в данной работе в качестве нерекурсивного.

Формула для нахождения расстояния с использованием матрицы:

(2)

**Расстояние Дамерау-Левенштейна.**

Между двумя строками a, b определяется функцией da,b(|a|,|b|) как:

(3)

Где – индикаторная функция, равная нулю при и 1 в противном случае.

Формула (3) выводится по тем же соображениям, что и формула (1).

Т. к. прямое применение этой формулы неэффективно, то аналогично действиям из предыдущего пункта производится добавление матрицы для хранения промежуточных значений рекурсивной формулы и оптимизация по памяти. В таком случае необходимо хранить одномерный массив длиной 3 min(|𝑎|, |𝑏|).

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.

## Конструкторская часть

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

### Разработка алгоритмов

Ниже будут приведены схемы реализованных алгоритмов:



Рисунок 1. Алгоритм Левенштейна часть 1



Рисунок 2. Алгоритм Левенштейна часть 2



Рисунок 3. Алгоритм Дамерау-Левенштейна часть 1



Рисунок 4. Алгоритм Дамерау-Левенштейна часть 2

L – Длина, S{n} – взятие подстроки от 0, до n – го символа строки S



Рисунок 5. Алгоритм Дамерау-Левенштейна рекурсивно часть 1



Рисунок 6. Алгоритм Дамерау-Левенштейна рекурсивно часть 2

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

Рекурсивный алгоритм, по сравнению с матричной реализацией работает медленнее, так как при исчерпывании всех возможных комбинаций, возникает ситуация, когда рекурсивные вызовы функций абсолютно идентичны и происходит нерациональная трата ресурсов.

Например:

В Рекурсивной реализации алгоритма Левенштейна при заданных словах «скат», «кот» будет выполнено 2 одинаковых рекурсивных вызова («ска», «ко»). Неполный фрагмент работы:



Рисунок 7. Минус рекурсивного алгоритма

Затраты в памяти реализаций алгоритмов так же отличаются:

Пусть n = s1.lenght(); m = s2.lenght();

Матричная реализация Левенштейна:

* (n + 1) \* ( m + 1) \* sizeof(int) + 6 \* int + 1 вызов функции min()

Матричная реализация Дамерау-Левенштейна:

* (n + 1) \* ( m + 1) \* sizeof(int) + 6 \* int + 2 вызова функции min()

Рекурсивная реализация Дамерау-Левенштрейна:

* 2 \* sizeof(int) + 2 вызова min() + 4 вызова самой себя

## 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. Среда разработки - Dev C/C++. Время работы процессора замеряется с помощью функции:

Листинг 1. Функция замера времени

unsigned long long **tick**(void)

{

unsigned long long d;

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

*return* d;

}

### Листинг кода

Листинг 2. Реализация Левенштейна с помощью матрицы

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 - 1] == s2[j - 1])

{

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;

}

Листинг 3. Реализация Дамерау-Левенштейна рекурсивно

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

{

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

*return* s2.length();

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

*return* s1.length();

int cost = 1;

*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() > 1 && s2.length() > 1) && 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);

}

*//unsigned* *tick2* *=* *tick();*

*//cout* *<<* *tick1* *<<* *"* *"* *<<* *tick2* *<<* *endl;*

*return* result;

}

Листинг 4. Реализация алгоритма Дамерау-Левенштейна

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 = 1;

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

{

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

{

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

{

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 - 1] == s2[j - 2] && s2[j - 1] == s1[i - 2])

{

matrix[i][j] = \_\_min(matrix[i][j], matrix[i - 2][j - 2] + cost);

}

}

}

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

clearMatrix(matrix, row\_count);

*return* result;

}

### Описание тестирования

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

1) Интоксикация, Детоксикация

2) Крот, Кот

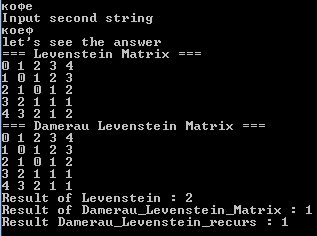
3) Кофе, Коеф

4) “ “, “ “  
5) “Кот”, “ Корт“

## Экспериментальная часть

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

### Примеры работы



Скриншот 1. Пример работы на словах кофе и коеф. Демонстрация разницы алгоритмов

### Результаты тестирования

Листинг 5. Функция тестирования

int **mainTest**(void)

{

string strings[10] = { "Intoxication", "Detoxication", "Krot", "Kot", "Cofe", "Coef", " ", " ", "Kot", "Kort"};

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

DamerauLevensteinRTesting(strings);

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

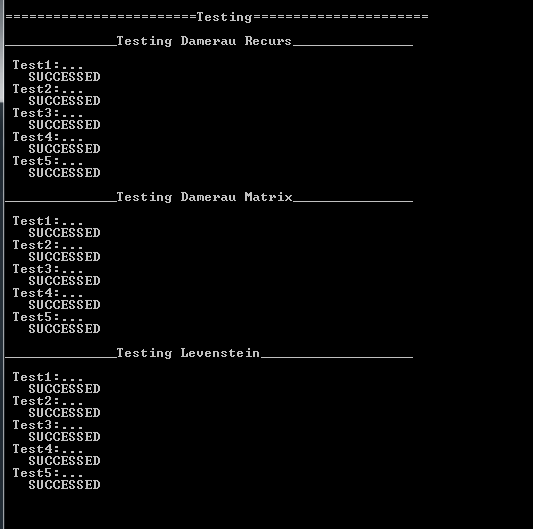
DamerauLevensteinMTesting(strings);

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

LevensteinMTesting(strings);

*return* 0;

}



*Скриншот 3. Результат тестирования*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № | S1 | S2 | Ожидаемый результат | Полученный результат |
| 1 | Пустая строка | Пустая строка | 0, 0, 0 | 0, 0, 0 |
| 2 | Пустая строка | ааааа | 5, 5, 5 | 5, 5, 5 |
| 3 | ааааа | Пустая строка | 5, 5, 5 | 5, 5, 5 |
| 4 | \s | \s | 0, 0, 0 | 0, 0, 0 |
| 5 | same | same | 0, 0, 0 | 0, 0, 0 |
| 6 | cofe | coef | 2, 1, 1 | 2, 1, 1 |
| 7 | krot | kot | 1, 1, 1 | 1, 1, 1 |
| 8 | kot | kort | 1, 1, 1 | 1, 1, 1 |
| 9 | spot | spok | 1, 1, 1 | 1, 1, 1 |

*Таблица 1. Результаты тестирования*

В таблице 1 в столбцах результатов указаны 3 значения через запятую, соответствующие реализациям алгоритмов Левенштейна, Дамерау-Левенштейна и рекурсивного алгоритма Дамерау-Левенштрейна. Во время тестирования программа корректно сработала на ввод пустых строк и строк содержащие одинаковые символы, а так же на операции удаления, добавления, замены и операции перестановки.

### 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.

Количество замеров будет взято равным 50.

Тестирование будет проведено на одинаковых входных данных. Для сравнения матричных реализации длины строк 0-500 с шагом 100. Для сравнения матричной и рекурсивной реализации длины строк 0-10 с шагом 1.

### Сравнительный анализ на материале экспериментальных данных

Ниже приведены графики зависимости временных затрат ( в тиках процессора) от размеров входных данных.

*График 1. Сравнение матричных реализаций*

По графику видно, что алгоритм Дамерау-Левенштейна работает медленнее алгоритма Левенштейна, так как в нем есть дополнительные проверки.

*График 2. Сравнение матричной и рекурсивной реализации алгоритма Дамерау-Левенштейна*

В результате проведенного эксперимента был получен следующий вывод:

Рекурсивный алгоритм Дамерау-Левенштейна работает гораздо дольше итеративной реализации, начиная с длины строк = 6, время его работы увеличивается в геометрической прогрессии. Итеративный алгоритм значительно превосходит его по эффективности.

# Заключение

В ходе работы были изучены и реализованы в матричной форме алгоритмы Левенштейна и Дамерау-Левенштейна и в рекурсивной алгоритм Дамерау-Левенштейна. Выполнено сравнение рекурсивного и итеративного алгоритмов Левенштейна. Изучены зависимости времени выполнения алгоритмов от длин строк. Также были реализованы 3 описанных алгоритма нахождения расстояний Левенштейна и Дамерау-Левенштейна.