|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **Министерство науки и высшего образования 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)** |

ФАКУЛЬТЕТ Информатика и системы управления (ИУ)

КАФЕДРА Программное обеспечение ЭВМ и информационные технологии (ИУ7)

**Отчет по лабораторной работе №1**

**По курсу: «Анализ алгоритмов»**

**«*Алгоритм по нахождению расстояния Левенштейна и Дамерау-Левенштейна*»**

Выполнил студент: \_\_***Тимонин Антон Срегеевич***\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*фамилия, имя, отчество*

Группа: \_\_\_\_***ИУ7-52Б***\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Преподаватель:.**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Волкова Л.Л.**

*подпись, дата*

Оценка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Дата \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*2019 г.*

**Оглавление**

[**Введение** 3](#_Toc22079556)

[**1 Аналитическая часть** 4](#_Toc22079557)

[**1.1 Расстояние Левенштейна** 4](#_Toc22079558)

[**1.2 Расстояние Дамерау-Левенштейна** 5](#_Toc22079559)

[**2 Конструкторская часть** 6](#_Toc22079560)

[**2.1 Требования к программному продукту** 6](#_Toc22079561)

[**2.2 Математическое описание** 6](#_Toc22079562)

[**2.3 Схемы алгоритмов** 7](#_Toc22079563)

[**2.4 Сравнительный анализ** 10](#_Toc22079564)

[**3 Технологическая часть** 11](#_Toc22079565)

[**3.1 Требования к ПО** 11](#_Toc22079566)

[**3.2 Листинг кода** 11](#_Toc22079567)

[**4 Экспериментальная часть** 14](#_Toc22079568)

[**4.1 Примеры работы программы** 14](#_Toc22079569)

[**4.2 Постановка эксперимента** 14](#_Toc22079570)

[**4.3 Сравнительный анализ** 16](#_Toc22079571)

[**5 Заключение** 18](#_Toc22079572)

# **Введение**

Целью данной лабораторной работы является изучение метода динамического программирования на материале алгоритмов Левенштейна и Левенштейна-Дамерау.

Задачами данной лабораторной работы являются:

1. реализация матричного способа для обоих расстояний и рекурсивного способа для расстояния Дамерау-Левенштейна;
2. применение метода динамического программирования для матричной реализации указанных алгоритмов;
3. получение практических навыков реализации указанных алгоритмов;
4. сравнительный анализ линейной и рекурсивной реализации выбранного алгоритма;
5. описание и обоснование полученных результатов в отчете о выполненной лабораторной работе.

# **1 Аналитическая часть**

В данном разделе приведены описания алгоритмов расстояние Левенштейна и Дамерау-Левенштейна

На вход алгоритма подаются две строки s1, s2 и длинами соответственно len1, len2. Тогда расстояние Левенштейна можно представить по рекуррентной формуле:

*D*(*s*1[1..*len*1], *s*2[1..*len*2]) = *min*(*D*(*s*1[1..*len*1 - 1], *s*2[1..*len*2] ) + 1,

*s*1[1..*len*1], *s*2[1..*len*2 - 1]+1,

*s*1[1..*len*1 - 1], *s*2[1..*len*2 - 1] + α),

Где α = 0, если s1[len1] = s2[len2], иначе α = 1

## **1.1 Расстояние Левенштейна**

Расстояние Левенштейна (*редакционное расстояние*,  *дистанция редактирования*) — минимальное количество операций вставки одного символа **I**, удаления одного символа **D** и замены одного символа на другой **R**, необходимых для превращения одной строки в другую. Измеряется для двух строк, часто используется в компьютерной лингвистике и биоинформатике, а в частности:

* для исправления ошибок в слове (в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи);
* для сравнения текстовых файлов утилитой diff и ей подобными. Здесь роль «символов» играют строки, а роль «строк» — файлы;
* в биоинформатике для сравнения генов, хромосом и белков.

В общем случае операции выглядят следующим образом:

D(a, b) – цена замены символа a на b

D(ø, b) – цена вставки символа b

D(a, ø) – цена удаления символа a

Также необходимо найти последовательность замен, минимизирующую суммарную цену:

Пусть a, b – символы; Ꜫ - пустой символ;

w(a, a) = 0;

w(a, b) = 1 при a≠b

w(a, Ꜫ) = 1

w(Ꜫ, a) = 1

## **1.2 Расстояние Дамерау-Левенштейна**

Расстояние Дамерау-Левенштейна – это мера разницы двух строк символов, определяемая как минимальное количество операций вставки, удаления, замены и транспозиции (перестановки двух соседних символов), необходимых для перевода одной строки в другую

По сути, это алгоритм является модификацией алгоритма расстояния Левенштейна, в который добавили операцию транспозиции.

# **2 Конструкторская часть**

В разделе конструкторская часть приведены требования к программе, математическое описание алгоритмов, блок схемы, сравнительный анализ

## **2.1 Требования к программному продукту**

Требования в вводу:

1. На вход подаются буквы латинского алфавита.

Требования к программе:

1. работа программы без аварийных завершений программы.

## **2.2 Математическое описание**

Классификация операций и штрафов (на выполнение операции) в алгоритме Левенштейна:

1. Вставка символа I (*insert) = 1*
2. Удаление символа D (*delete*) = 1
3. Замена символа R (*replace*) = 1
4. Совпадение символа M (*match*) = 0

В алгоритме Дамерау-Левенштейна добавляется операция транспозиции, на случай если 2 символа стоят не в том порядке:

1. Транспозиция T (*transposition*) = 1

А также в рекуррентную формула добавляется еще один член:

*D*(*s*1[1..*len*1], *s*2[1..*len*2 - 1]) + β,

где β = 1, если *s*1[*len*1 – 1] = *s*2[*len*2] и *s*2[*len*2 - 1] = s1[*len*1], а иначе β = 0

## **2.3 Схемы алгоритмов**

Матричная реализация алгоритма нахождения расстояния Левенштейна приведена на рис. 1.

Ввод строк s1 и s2

Инициализация матрицы matr[len(s1)+1][len(s2)+1]

Цикл1 от 1 до len(s1)

alph = 0

alph = 1

s1[i-1] = s2[j-1]

Цикл2 от 1 до len(s2)

Цикл1 от 1 до len(s1)

Matr[i][j] = min( matr[i-1][j]+1, matr[i][j-1]+1, matr[i-1][j-1]+ alph)

Цикл2 от 1 до len(s2)

Начало

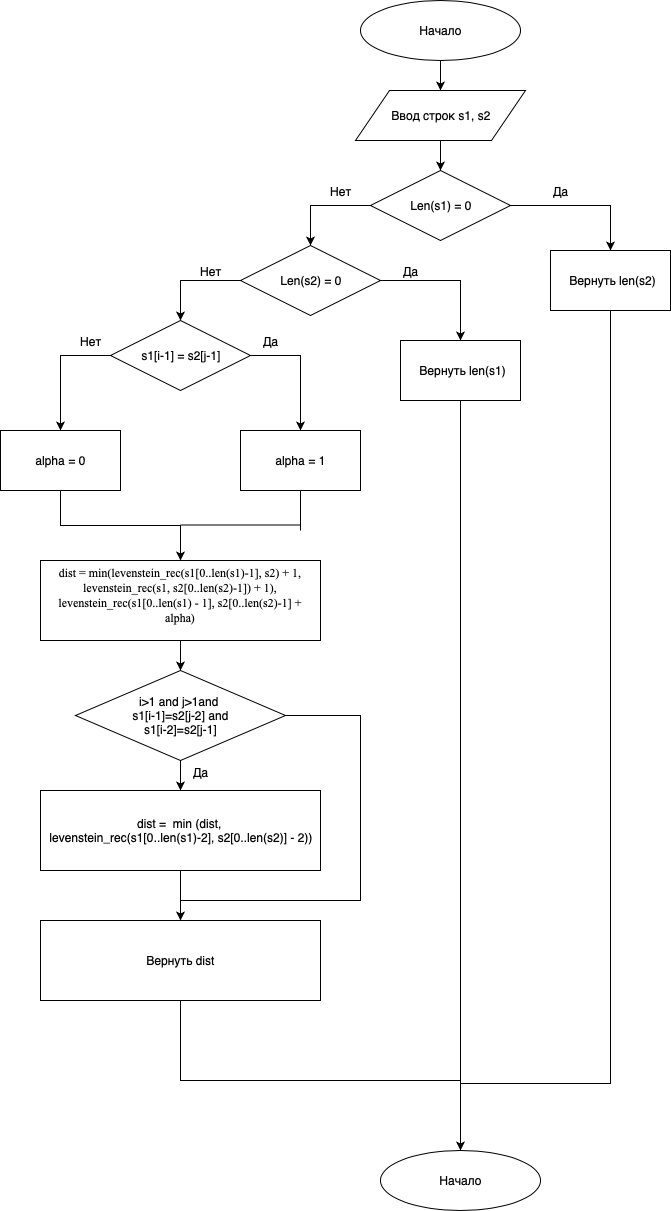
Конец

Да

Нет

Рис. 1 – Название

Рекурсивная реализация алгоритма нахождения расстояния Дамерау-Левенштейна (Рис. 2)



Матричная реализация алгоритма Дамерау-Левенштейна (Рис. 3)

Ввод строк s1 и s2

Инициализация матрицы matr[len(s1)+1][len(s2)+1]

Цикл1 от 1 до len(s1)

alph = 0

alph = 1

s1[i-1] = s2[j-1]

Цикл2 от 1 до len(s2)

Цикл1 от 1 до len(s1)

Matr[i][j] = min( matr[i-1][j]+1, matr[i][j-1]+1, matr[i-1][j-1]+ alph)

Цикл2 от 1 до len(s2)

Начало

Конец

i>1 и j>1 и s1[i-1]=s2[j-2] и s1[i-2]=s2[j-1]

нет

Matr[i][j] = min(. matr[i][j], matr[i-2]j-2)+1)

## **2.4 Сравнительный анализ**

При больших входных данных рекурсивная реализация занимает значительно больше тиков процессор, чем обычная, матричная реализация. Но в некоторых случаях, когда длина слов, подающихся на вход, мала, тогда рекурсивная реализация занимает меньше памяти.

# **3 Технологическая часть**

В данном разделе приведено требования к программному обеспечению, а также листинг кода

## **3.1 Требования к ПО**

Программа была написана на языке С++ в среде QtCreator. Данный язык обусловлен наличием библиотеки, позволяющей считать тики процессора, что более точно определяет эффективность программы

## **3.2 Листинг кода**

Матричная реализация алгоритма Левенштейна

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

{

int len\_s1 = s1.length() + 1;

int len\_s2 = s2.length() + 1;

int arr[len\_s1][len\_s2];

for (int i = 0; i < len\_s1; i++) {

for (int j = 0; j < len\_s2; j++) {

if (i \* j == 0) {

arr[i][j] = i + j;

} else {

arr[i][j] = 0;

}

cout << arr[i][j] << " ";

}

cout << "\n";

}

for (int i = 1; i < len\_s1; i++) {

for (int j = 1; j < len\_s2; j++) {

int key = 1;

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

key = 0;

}

arr[i][j] = Mmin(arr[i-1][j]+1, arr[i][j-1]+1,

arr[i-1][j-1]+key);

}

}

return arr[len\_s1 - 1][len\_s2 - 1];

}

Матричная реализация алгоритма Дамерау-Левенштейна

int **dameray\_levenstein**(string s1, string s2)

{

int len\_s1 = s1.length() + 1;

int len\_s2 = s2.length() + 1;

int key = 1;

int arr[len\_s1][len\_s2];

for (int i = 0; i < len\_s1; i++) {

for (int j = 0; j < len\_s2; j++) {

if (i \* j == 0) {

arr[i][j] = i + j;

} else {

arr[i][j] = 0;

}

}

}

for (int i = 1; i < len\_s1; i++) {

for (int j = 1; j < len\_s2; j++) {

key = 1;

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

key = 0;

arr[i][j] = Mmin(arr[i-1][j] + 1, arr[i][j-1] + 1,

arr[i-1][j-1] + key);

if (i > 1 && j > 1 && s1[i-1] == s2[j-2] && s1[i-2] == s2[j-1]) {

arr[i][j] = Mmin(arr[i][j], arr[i][j], arr[i-2][j-2] + 1);

}

}

}

return arr[len\_s1 - 1][len\_s2 - 1];

}

Рекурсивная реализация алгоритма расстояние Дамерау-Левенштейна

int **dameray\_levenstein\_rec**(string s1, string s2)

{

int var = 1;

int dist = 0;

int s1\_l, s2\_l;

s1\_l = Llen(s1);

s2\_l = Llen(s2);

if (s1 == "" || s2 == "") {

dist = max(s1.length(), s2.length());

return dist;

}

string s1\_new = s1.substr(0, s1.length() - 1);

string s2\_new = s2.substr(0, s2.length() - 1);

if (s1[s1\_l] == s2[s2\_l]) {

var = 0;

}

dist = Mmin (dameray\_levenstein\_rec(s1\_new, s2) + 1,

dameray\_levenstein\_rec(s1, s2\_new) + 1,

dameray\_levenstein\_rec(s1\_new, s2\_new) + var);

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

s1[s1\_l - 1] == s2[s2\_l]) {

string s1\_damer = s1.substr(0, s1.length() - 2);

string s2\_damer = s2.substr(0, s2.length() - 2);

dist = std::min(dist,dameray\_levenstein\_rec(s1\_damer, s2\_damer) + 1);

}

return dist;

}

# **4 Экспериментальная часть**

## **4.1 Примеры работы программы**

Пример работы программы представлены в таблицах 1.1 и 2.2. В 3-4 столбцах приведены результирующие расстояния, алгоритмы идут в порядке: матричный алгоритм расстояние Левенштейна, рекурсивный алгоритм Левенштейна, матричный алгоритм Дамерау-Левенштейна.

Таблица 1.1

|  |  |  |  |
| --- | --- | --- | --- |
| Word 1 | Word 2 | Output | Expected output |
| mainframe | aminframe | 2 1 1 | 2 1 1 |
| mainframe | mainfrem | 9 8 8 | 9 8 8 |
| mainframe |  | 9 9 9 | 9 9 9 |
| mainframe | mainframe | 0 0 0 | 0 0 0 |

Таблица 1.2

|  |  |  |  |
| --- | --- | --- | --- |
| Word 1 | Word 2 | Output | Expected output |
| abcdef | badcfe | 4 3 3 | 4 3 3 |
| abcdef | abcfed | 2 2 2 | 2 2 2 |
| abcdef | cbed | 4 4 4 | 4 4 4 |
| abcdef | fedcba | 6 5 5 | 6 5 5 |

## **4.2 Постановка эксперимента**

Оценка эффективности алгоритмов по скорости работы представлена на Рис 4 и Рис 5

Рис. 4 – Эффективность по времени работы в зависимости от входных слов длиной до 10

Рисунок 5: Эффективность по времени работы в зависимости от входных слов длиной до 6

Если брать больше 6 слов, тогда диаграмма масштабируется так, что видно только самое большое значение в рекурсивной вариации алгоритма Левенштейна.

|  |  |  |  |
| --- | --- | --- | --- |
| Кол-во слов | Матр. Левенштейн | Матр. Дамерау-Левенштейн | Рекурс. Дамерау-Левенштейн |
| 0 | 3621 | 1989 | 2228 |
| 2 | 373477 | 286710 | 7600 |
| 4 | 395128 | 312099 | 195000 |
| 6 | 377175 | 229074 | 3214622 |
| 8 | 825228 | 592711 | 90750156 |
| 10 | 512247 | 237957 | 1592646320 |

Таблица 2.1: Эффективность алгоритмов по тикам процессора

|  |  |  |  |
| --- | --- | --- | --- |
| Длина строки | Левенштейн | Дамер.Левеншт. | Дамер.Левеншт.Рекурс. |
| 0 | 8 | 12 | 0 |
| 2 | 24 | 28 | 120 |
| 4 | 72 | 76 | 551 |
| 6 | 152 | 156 | 2615 |
| 8 | 264 | 268 | 14232 |
| 10 | 408 | 412 | 79384 |
|  |  |  |  |

Таблица 2.2: Эффективность алгоритмов по памяти в байтах

Рисунок 6: Эффективность по памяти в зависимости от входных слов длиной до 10

## **4.3 Сравнительный анализ**

По данным эксперимента видно, что при больших входных данных скорость работы рекурсивной реализации алгоритма по нахождению расстояния Левенштейна сильно уменьшается в сравнении с матричной реализацией данного алгоритма. Также при больших входных данных могут возникать сбои программы.

Также алгоритм Дамерау-Левенштейна предпочтительней, в связи с тем, что в данном алгоритме присутствует операция транспозиции. Но если брать во внимание рекурсивную реализацию алгоритма Дамерау-Левенштейна, то будет наблюдаться такой же экспоненциальный рост времени работы, поэтому матричная реализация более целесообразна.

Таким образом, для проблемы исправления ошибок лучше всего взять во внимание матричную реализацию алгоритма Дамерау-Левенштейна

# **5 Заключение**

В данной лабораторной работы были исследованы 3 реализации алгоритмов для поиска редакционного расстояния. После проведения некоторого количества экспериментов, был произведен сравнительный анализ, на основе которого была выявлена эффективность матричной реализации двух вариаций алгоритмов.

Проведя экспериментальную часть, оказалось, что матричная(итеративная) реализация алгоритмов оказалась намного эффективнее по памяти и по скорости работы, в отличие от рекурсивной. Однако, рекурсивная реализация оказалась проще по написанию на языке программирования С++, что было выявлено в технологической части.