Государственное образовательное учреждение высшего профессионального образования
“Московский государственный технический университет имени Н.Э.Баумана”

Дисциплина: Анализ алгоритмов

Лабораторная работа №1

Расстояние Левенштейна

Студент группы ИУ7-54Б,
Котов Никита

2019 г.

Содержание

Введение

Расстояние Левенштейна определяет, сколько раз необходимо добавить/удалить/заменить символ, чтобы одну строку превратить в другую.

Впервые задачу упомянул в 1965 году советский математик Владимир Иосифович Левенштейн при изучении последовательностей[1]. Впоследствии более общую задачу для произвольного алфавита связали с его именем. Позже большой вклад в изучение вопроса внёс Дэн Гасфилд.

Расстояние Левенштейна может играть роль фильтра, заведомо отбрасывающего неприемлемые варианты, у которых значение функции больше некоторой заданной константы.

Так же существует понятие расстояния Дамерау-Левенштейна. Его определение аналогично расстоянию Левенштейна, но добавляется операция транспозиции (перестановки двух соседних символов).

Расстояние Левенштейна и его обобщения активно применяется:

- для исправления ошибок в слове[2] (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[3];
- в биоинформатике для сравнения генов, хромосом и белков[4].

В рамках выполнения работы необходимо решить следующие задачи:

- рассмотреть и изучить понятия расстояния Левенштейна и расстояния Дамерау-Левенштейна;
- реализовать два варианта алгоритма нахождения расстояния Левенштейна (рекурсивного и нерекурсивного вида);
- сравнить их временные характеристики экспериментально;
- реализовать алгоритм нахождения расстояния Дамерау-Левенштейна
- на основании проделанной работы сделать выводы.

1 Аналитическая часть

1.1 Описание алгоритмов

Рекурсивный алгоритм нахождения расстояния Левенштейна

Расстояние Левенштейна между двумя строками a и b может быть вычис лено по формуле D(|a|, |b|), где |a| означает длину строки a; a[i] — i-ый символ строки a , функция D(i, j) определена как:

$$D(i,j) = \begin{cases} 0 & \text{i} = 0, \text{j} = 0 \\ i & \text{j} = 0, \text{i} > 0 \\ j & \text{i} = 0, \text{j} > 0 \\ min\{ & \\ \quad D(i,j-1)+1 & \\ \quad D(i-1,j)+1 & \text{i} > 0, \text{j} > 0 \\ \quad D(i-1,j-1)+m(a[i],b[j]) & \\ \} & \end{cases} \tag{1}$$

a функция m(a, b) определена как

$$m(a,b) = \begin{cases} 0 & \text{если a} = \text{b} \\ 1 & \text{иначе} \end{cases} \tag{2}$$

Рекурсивный алгоритм реализует данную формулу. Функция D 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):
   - сумма цены преобразования строки a в b и цены проведения операции удаления, которая необходима для преобразования a' в a ;
   - сумма цены преобразования строки a в b и цены проведения операции вставки, которая необходима для преобразования b' в b ;
   - сумма цены преобразования из a' в b' и операции замены, предполагая, что a и b оканчиваются разные символы;
   - 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 приведенной выше формулы D может быть малоэффективна при больших i, j, т. к. множество промежуточных значений D(i, j) вычисляются заново множество раз подряд. Для оптимизации нахождения расстояния Левенштейна можно использовать матрицу в целях хранения соответствующих промежуточных значений. В таком случае алгоритм представляет собой построчное заполнение матрицы A|a|,|b| значениями D(i, j).

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 |b|. Такой вариант алгоритма называется построчным и именно он реализован в данной работе в качестве нерекурсивного.

Нахождение расстояния Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна может быть найдено по рекурсивной формуле $\mathrm{d_{a,b}}(|a|,|b|)$, где $\mathrm{d_{a,b}}(i, j)$ задана как

$$d_{\mathrm{a,b}}(i,j) = \begin{cases} max(i,j) & \text{если } \min(i,j) = 0, \\ min\{ & \\ \quad d_{\mathrm{a,b}}(i,j-1)+1 & \\ \quad d_{\mathrm{a,b}}(i-1,j)+1 & \text{иначе} \\ \quad d_{\mathrm{a,b}}(i-1,j-1)+m(a[i],b[j]) & \\ \quad M_{\mathrm{a,b}}(i,j) & \\ \} & \end{cases} \tag{3}$$

где $\mathrm{M_{a,b}}(i, j)$ задается как

$$M_{\mathrm{a,b}}(i,j) = \begin{cases} d_{\mathrm{a,b}}(i-2,j-2)+1 & \text{если i,j} > 1;\ \text{a[i]} = \text{b[j-1]};\ \text{b[j]} = \text{a[i-1]} \\ +\infty & \text{иначе} \end{cases} \tag{4}$$

Формула 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|) .

2 Конструкторская часть

2.1 Разработка алгоритмов

2.1.1 Рекурсивный алгоритм вычисления расстояния Левенштейна

На рис. 1 представлена схема рекурсивного алгоритма нахождения расстояния Левенштейна

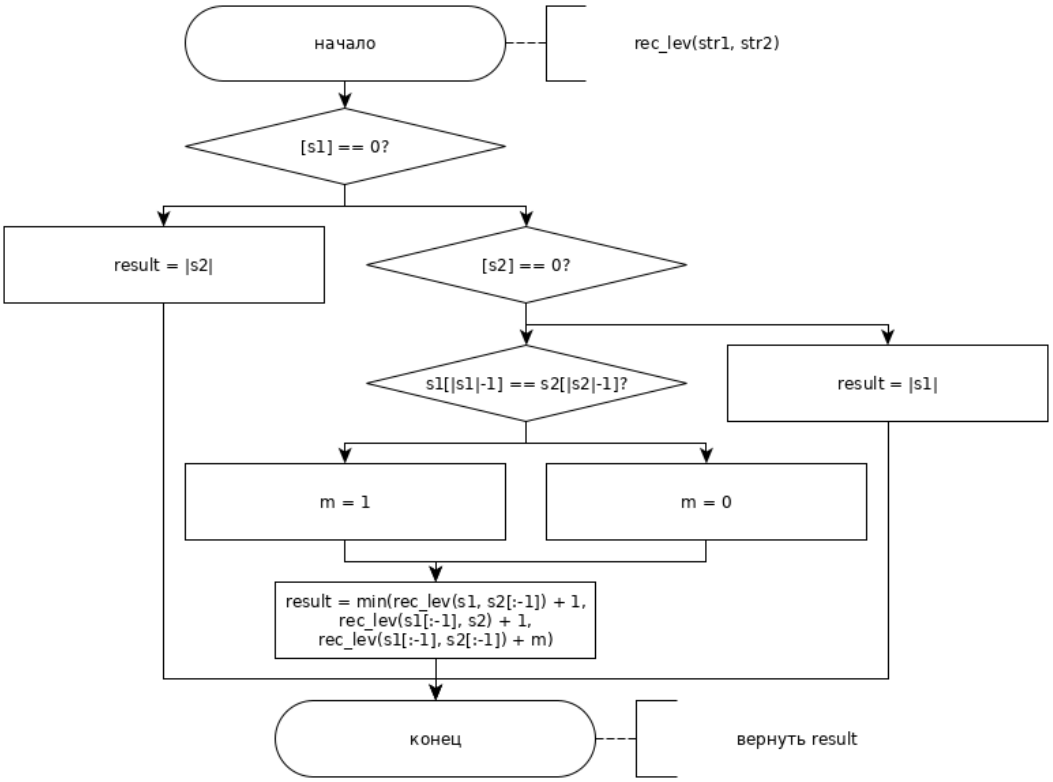

Рис. 1: Рекурсивный алгоритм вычисления расстояния Левенштейна

Алгоритм 1 Рекурсивный алгоритм вычисления расстояния Левенштейна $dist(s, t)$

if $|s| = 0$ **then**
  return $|t|$
if $|t| = 0$ **then**
  return $|s|$
$val \leftarrow 1$
if $s[|s|] = t[|t|]$ **then**
  $val \leftarrow 0$
return $min \begin{cases} dist(s[1 \dots |s| - 1], t) + 1 \\ dist(s, t[1 \dots |t| - 1]) + 1 \\ dist(s[1 \dots |s| - 1], t[1 \dots |t| - 1]) + val \end{cases}$

2.1.2 Матричный алгоритм вычисления расстояния Левенштейна

На рис. 2 и рис. 3 представлена схема матричного алгоритма нахождения расстояния Левенштейна

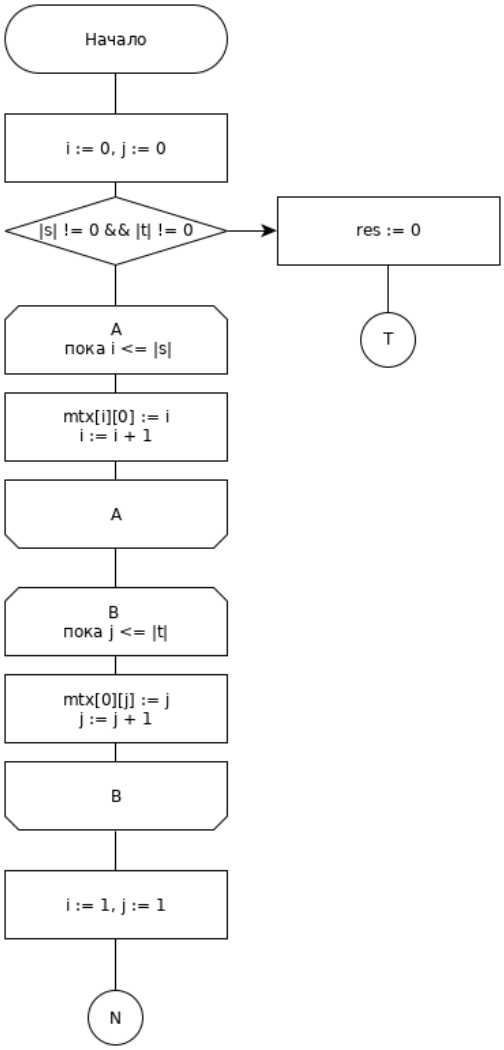

Рис. 2: Матричный алгоритм вычисления расстояния Левенштейна

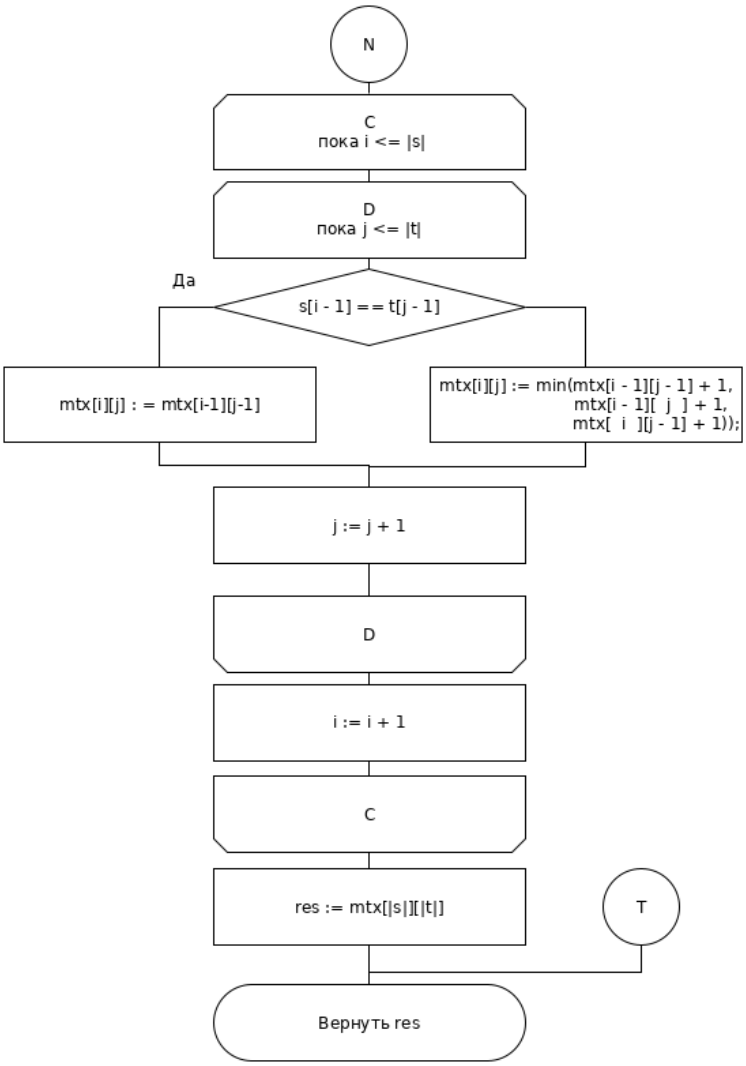

Рис. 3: Матричный алгоритм вычисления расстояния Левенштейна

Алгоритм 2 Матричный алгоритм вычисления расстояния Левенштейна $dist(s, t)$

$mtx[0 \dots n, 0 \dots m] \leftarrow 0$
for от $i = 0$ **из** $[0 \dots \mathrm{n}]$ **do**
    $mtx[0, i] \leftarrow i$
for от $i = 1$ **из** $[1 \dots \mathrm{n}]$ **do**
    $mtx[i, 0] \leftarrow i$

for от $i = 0$ **из** $[0 \dots \mathrm{n}]$ **do**
    for от $j = 0$ **из** $[0 \dots \mathrm{m}]$ **do**
        if $s[i] = t[j]$ **then**
            $mtx[i, j] = mtx[i - 1, j - 1]$
        else
            $mtx[i, j] = min(mtx[i, j - 1] + 1, mtx[i - 1][j] + 1, mtx[i - 1, j - 1] + 1)$
return $mtx[m, n]$

2.1.3 Матричный алгоритм вычисления расстояния Левенштейна

На рис. 4 и рис. 5 представлена схема матричного алгоритма нахождения расстояния Левенштейна

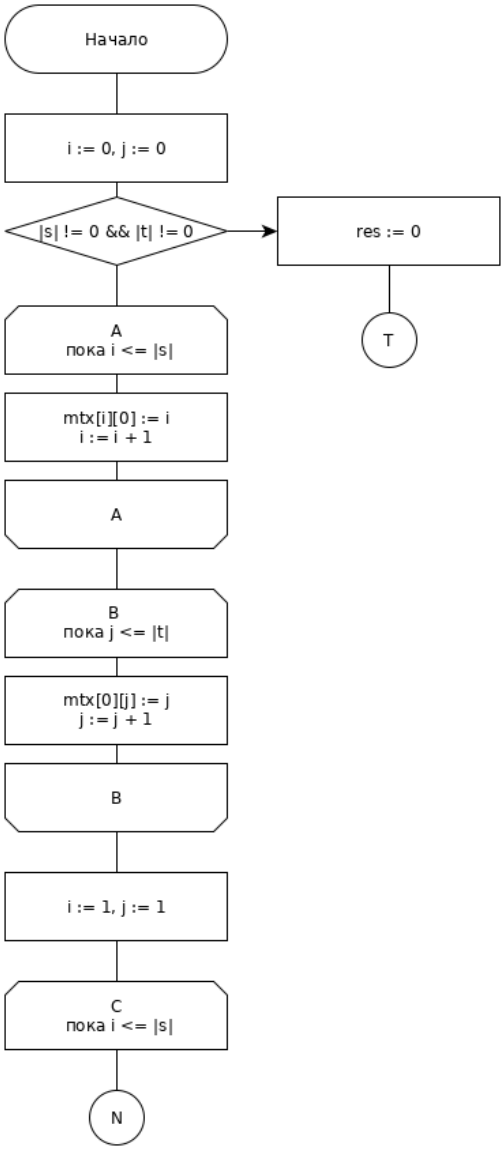

Рис. 4: Матричный алгоритм вычисления расстояния Дамерау-Левенштейна

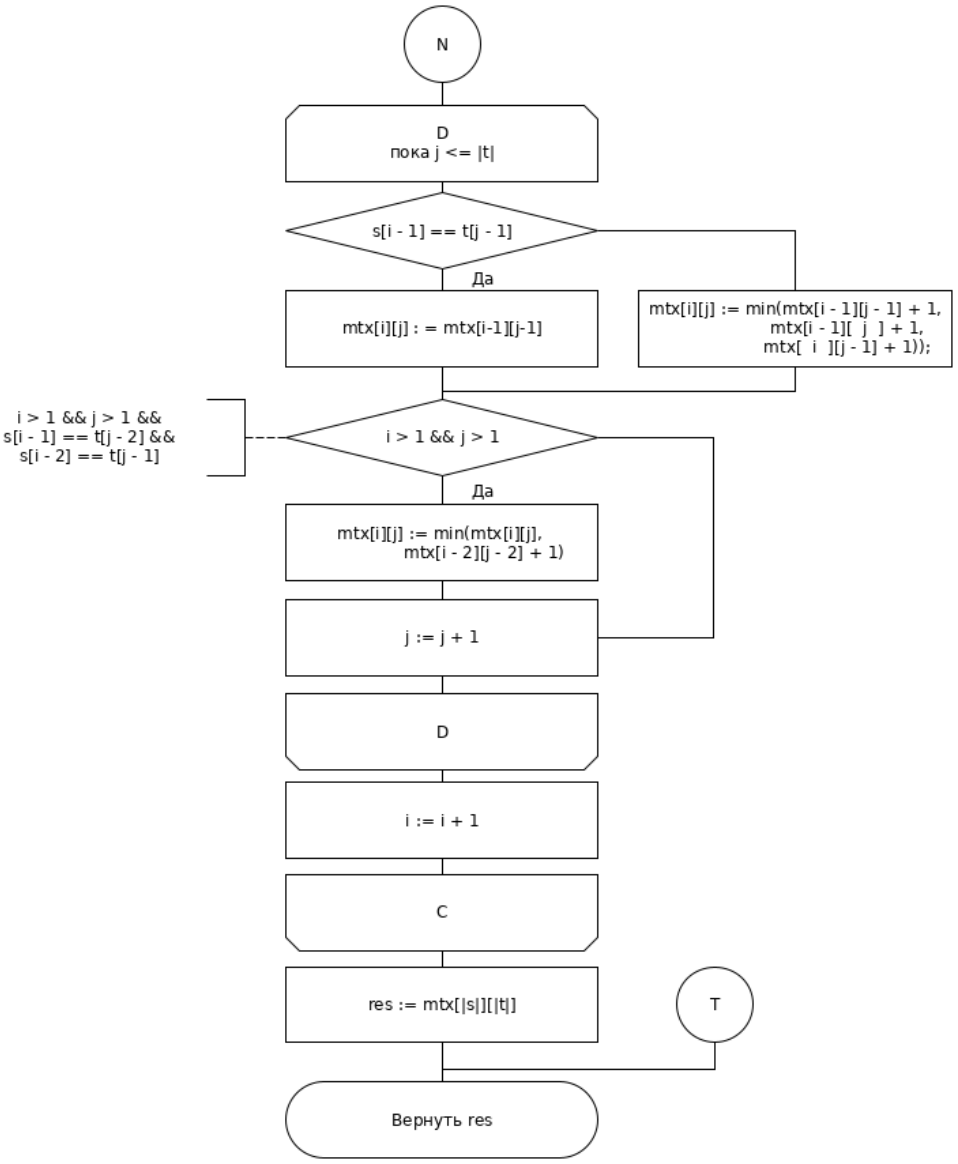

Рис. 5: Матричный алгоритм вычисления расстояния Дамерау-Левенштейна

Алгоритм 3 Матричный алгоритм вычисления расстояния Дамерау-Левенштейна $dist(s, t)$

$mtx[0 \dots n, 0 \dots m] \leftarrow 0$
for от $i = 0$ **из** $[0 \dots \mathrm{n}]$ **do**
    $mtx[0, i] \leftarrow i$
for от $i = 1$ **из** $[1 \dots \mathrm{n}]$ **do**
    $mtx[i, 0] \leftarrow i$

for от $i = 0$ **из** $[0 \dots \mathrm{n}]$ **do**
    for от $j = 0$ **из** $[0 \dots \mathrm{m}]$ **do**
        if $s[i] = t[j]$ **then**
            $mtx[i, j] = mtx[i - 1, j - 1]$
        else
            $mtx[i, j] = min(mtx[i, j - 1] + 1, mtx[i - 1][j] + 1, mtx[i - 1, j - 1] + 1)$
        if $i > 1 j > 1 s[i - 1] = t[j - 2] s[i - 2] = t[j - 1]$ **then**
            $mtx[i, j] = min(mtx[i, j], mtx[i - 2, j - 2] + 1)$
return $mtx[m, n]$

2.2 Выводы по конструкторскому разделу

Были разработаны схемы трех алгоритмов вычисления расстояния Левенштейна и его модификации, алгоритма Дамерау-Левенштейна. Каждый из них был описан с помощью псевдокода.

3 Технологическая часть

3.1 Требования к программному обеспечению

Программа должна работать в одном из двух режимов: пользовательском и эксперементальном.
В пользовательском режиме должны быть реализованы:

- ввод исходных данных с клавиатуры
- вывод результата работы алгоритмов
- отсутствие аварийных ситуаций

Эксперементальный режим должен также предоставлять сведенья о затраченном на выполнение алгоритмов процессорном времени.

3.2 Средства реализации

В качестве языка программирования был выбран C++, так как он предоставляет широкие возможности для эффективной реализации алгоритмов.

3.3 Структура программы

levenshtain_dist.cpp - содержит реализацию алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна
test.cpp - содержит тесты для реализованных алгоритмов
main.cpp - содержит интерфейс для взаимодействия с программой

3.4 Листинг кода

Листинг 1: Рекурсивный алгоритм вычисления расстояния Левенштейна

```
int recursive_lev(std::string str1, std::string str2) {
    auto len1 = str1.length();
    auto len2 = str2.length();

    if (len1 == 0) {
        return static_cast<int>(len2);
    }
    if (len2 == 0) {
        return static_cast<int>(len1);
    }

    auto last_symb = str1[len1 - 1] == str2[len2 - 1]? 0: 1;

    auto s1 = std::string(str1, 0, len1 - 1);
    auto s2 = std::string(str2, 0, len2 - 1);

    return std::min( { recursive_lev(str1, s2) + 1,
                       recursive_lev(s1, str2) + 1,
                       recursive_lev(s1, s2) + last_symb });
}
```

Листинг 2: Матричный алгоритм вычисления расстояния Левенштейна

```
int lev(std::string str1, std::string str2) {
    std::vector<int> a1(str2.length() + 1);
    std::vector<int> a2(str2.length() + 1);
    // initialisation of 1st row
    for (int i = 0; i < a1.size(); i++) a1[i] = i;

    for (int i = 1; i < str1.length() + 1; i++) {
        a2.at(0) = a1.at(0) + 1;
        for (int j = 1; j < str2.length() + 1; j++) {
            auto vertical = a1[j] + 1;
            auto horisontal = a2[j - 1] + 1;
            auto diagonal = a1[j - 1] + (str1[i - 1] == str2[j - 1]? 0: 1);
            a2[j] = std::min({ vertical, horisontal, diagonal });
        }
        std::copy(a2.begin(), a2.end(), a1.begin());
    }

    return a2[a2.size() - 1];
}
```

Листинг 3: Матричный алгоритм вычисления расстояния Дамерау-Левенштейна

```
int dam_lev_dist(std::string str1, std::string str2) {
    std::vector<int> a0(str2.length() + 1);
    std::vector<int> a1(str2.length() + 1);
    std::vector<int> a2(str2.length() + 1);
    // initialisation of 1st row
    for (int i = 0; i < a1.size(); i++) a1[i] = i;

    for (int i = 1; i < str1.length() + 1; i++) {
        a2[0] = a1[0] + 1;
        for (int j = 1; j < str2.length() + 1; j++) {
            auto eq = str1[i - 1] == str2[j - 1]? 0: 1;

            auto vertical = a1[j] + 1;
            auto horisontal = a2[j - 1] + 1;
            auto diagonal = a1[j - 1] + eq;
            a2[j] = std::min({ vertical, horisontal, diagonal });

            // swap
            if (i && j && str1[i - 1] == str2[j - 2]
             && str1[i - 2] == str2[j - 1]) {
                a2[j] = std::min(a2[j], a0[j - 2] + eq);
            }
        }
        a0 = a1;
        a1 = a2;
    }

    return a2[a2.size() - 1];
}
```

3.5 Тестирование фунций

В таблице 1, таблице 2, таблице 3 приведены результаты тестирования для рекурсивного, матричного алгоритмов вычисления расстояния Левенштейна и алгоритма нахождения расстояния Дамерау-Левенштейна соответственно.

Таблица 1: Рекурсивный алгоритм вычисления расстояния Левенштейна

Строка 1	Строка 2	Результат	Ожидаемый результат
kot	skat	2	2
abc	abc	0	0
kot	null	3	3
null	kot	3	3
null	null	0	0
abc	acb	2	2

Таблица 2: Матричный алгоритм вычисления расстояния Левенштейна

Строка 1	Строка 2	Результат	Ожидаемый результат
kot	skat	2	2
abc	abc	0	0
kot	null	3	3
null	kot	3	3
null	null	0	0
abc	acb	2	2

Таблица 3: Матричный алгоритм вычисления расстояния Дамерау-Левенштейна

Строка 1	Строка 2	Результат	Ожидаемый результат
kot	skat	2	2
abc	abc	0	0
kot	null	3	3
null	kot	3	3
null	null	0	0
abc	acb	1	1

4 Экспериментальная часть

4.1 Примеры работы

Входные данные:
Enter 1st string: **kot**
Enter 2nd string: **skat**
Run time tests? y /n? **n**

Выходные данные:
Levenshtein distance (Matrix)
Matrix:
- λ q w e
λ 0 1 2 3
a 1 1 2 3
d 2 2 2 3
e 3 3 3 3

Result: 3

Levenshtein distance (Recursive)
Result: 3

Damerau-Levenshtein distance (Matrix)
Matrix:
- λ q w e
λ 0 1 2 3
a 1 1 2 3
d 2 2 2 3
e 3 3 3 3

Result: 3

Входные данные:
Enter 1st string: **kot**
Enter 2nd string: **skat**
Run time tests? y /n? **y**

Выходные данные:
Levenshtein distance (Matrix)
Time: 0.00001110s
Matrix:
- λ q w e
λ 0 1 2 3
a 1 1 2 3
d 2 2 2 3
e 3 3 3 3

Result: 3

Levenshtein distance (Recursive)

Time: 0.00004630s
Result: 3

Damerau-Levenshtein distance (Matrix)
Time: 0.00001550s
Matrix:
- λ q w e
λ 0 1 2 3
a 1 1 2 3
d 2 2 2 3
e 3 3 3 3

Result: 3

4.2 Тестирование времени работы функций

Для измерения времени использовалась функция clock(). Чтобы исключить случайные отклонения в измеренном времени, измерялось время работы 50 запусков функции и делилось на 50.

В таблице 4 представлены результаты замеров времени работы реализованных алгоритмов. На рис. 6 и рис. 7 изображены графики, позволяющие сравнить эффективность реализаций.

Таблица 4: Время работы реализаций алгоритмов

|Строка 1|	|Строка 2|	МДЛ	МЛ	РЛ
200	200	0.002242	0.002173	-
100	100	0.000867	0.000848	-
50	50	0.000545	0.000607	-
10	10	0.000045	0.000041	0.430871
8	8	0.000031	0.000028	0.017353
5	5	0.000018	0.000015	0.000689
50	2	0.000042	0.000048	0.001515
50	4	0.000123	0.000114	0.283745

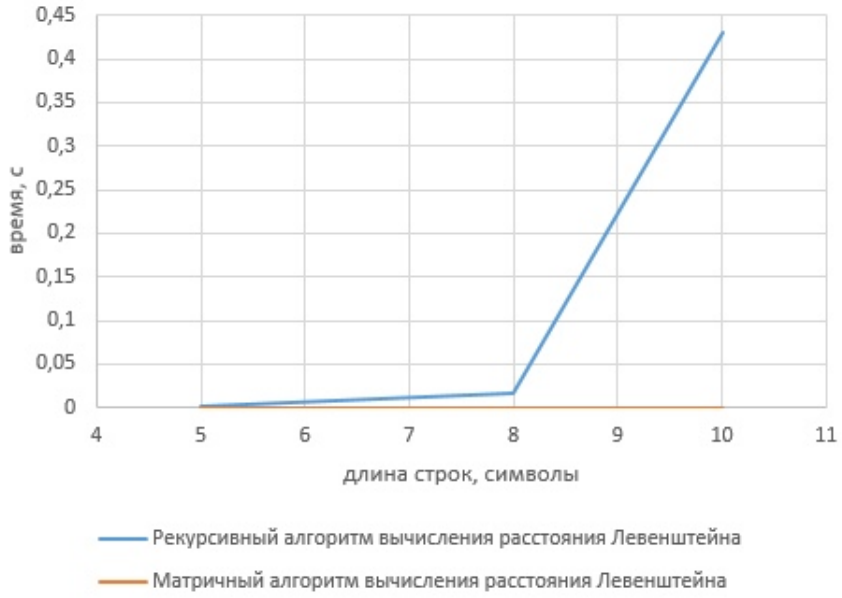

Рис. 6: Сравнение матричных реализаций алгоритмов

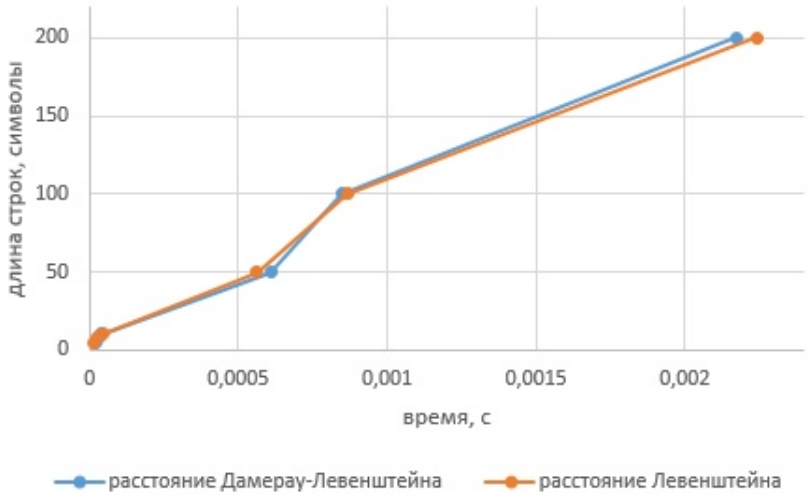

Рис. 7: Сравнение рекурсивного и матричного алгоритма

Время работы матричных реализаций практически не отличается друг от друга, а рекурсивная реализация алгоритма нахождения реализации Левенштейна уже на строках длины 9 работает медленнее в 500 раз.

Заключение

В рамках лабораторной работы были реализованы и изучены рекурсивный алгоритм нахождения расстояния Левенштейна, матричная версия алгоритма и алгоритм нахождения расстония Дамерау-Левенштейна. Были произведены замеры времени работы реализованных алгоритмов и сравнена их эффективность по времени и памяти. Матричные алгоритмы оказались близки по эффективности, рекурсивный алгоритм же работает значительно медленнее матричных реализаций, уже при длинне строк больше 10 установить время работы не удалось.

Список литературы

[1] В. И. Левенштейн. Двоичные коды с исправлением выпадений, вставок и замещений символов. Доклады Академий Наук СССР, 1965. 163.4:845-848.

[2] Damareau , F., J., "A technique for computer detection and correction of spelling errors"

[3] Ovicic, V., "Constrained edit distance algorithm and its application in Library Information Systems"

[4] The Institute of Electronics, "Tree Edit Distance Problems: Algorithms and Applications to Bioinformatics"

[5] Гасфилд. Строки, деревья и последовательности в алгоритмах. Информатика и вычислительная биология. Невский Диалект БВХ-Петербург, 2003.

[6] US National Library of Medicine National Institutes of Health, "Secure approximation of edit distance on genomic data"