МГТУ им. Баумана

Лабораторная работа №1

По курсу: "Анализ алгоритмов"

# Расстояние Левенштейна

Работу выполнил: Мокеев Даниил, ИУ7-54

Преподаватели: Волкова Л.Л., Строганов Ю.В.

*Москва, 2019*

# Оглавление

# Введение

**Расстояние Левенштейна** - минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

Расстояние Левенштейна применяется в теории информации и компьютерной лингвистике для:

- исправления ошибок в слове
- сравнения текстовых файлов утилитой diff и ей подобными
- в биоинформатике для сравнения генов, хромосом и белков

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

Задачами данной лабораторной являются:

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 временно́й 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

Задача по нахождению расстояния Левенштейна заключается в поиске минимального количества операций вставки/удаления/замены для превращения одной строки в другую.

При нахождении расстояния Дамерау — Левенштейна добавляется операция транспозиции (перестановки соседних символов).

**Действия обозначаются так:**

1. D (англ. delete) — удалить,

2. I (англ. insert) — вставить,

3. R (replace) — заменить,

4. M(match) - совпадение.

Пусть $S_1$ и $S_2$ — две строки (длиной M и N соответственно) над некоторым алфавитом, тогда расстояние Левенштейна можно подсчитать по следующей рекуррентной формуле:

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ min( & \\ D(i, j-1) + 1, & \\ D(i-1, j) + 1, & j > 0, i > 0 \\ D(i-1, j-1) + m(S_1[i], S_2[j]) & \\ ), & \end{cases}$$

где $m(a, b)$ равна нулю, если $a = b$ и единице в противном случае; $min\{\, a, b, c\}$ возвращает наименьший из аргументов.

Расстояние Дамерау-Левенштейна вычисляется по следующей рекуррентной формуле:

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ min( & \\ D(i, j-1) + 1, & \\ D(i-1, j) + 1, & j > 0, i > 0 \\ D(i-1, j-1) + m(S_1[i], S_2[j]) & \\ D(i-2, j-2) + 1, & \text{if } i, j > 1 \text{ and } a_i = b_{j-1}, a_{i-1} = b_j \\ ) & \end{cases}$$

# 2 Конструкторская часть

**Требования к вводу:**

1. На вход подаются две строки
2. uppercase и lowercase буквы считаются разными

**Требования к программе:**

1. Две пустые строки - корректный ввод, программа не должна аварийно завершаться

# 3 Технологическая часть

## 3.1 Выбор ЯП

Я выбрал в качестве Python языком программирования, потому как он достаточно удобен и гибок.

Время работы алгоритмов было замерено с помощью функции time() из библиотеки time.

## 3.2 Сведения о модулях программы

Программа состоит из:

- lab01.py - главный файл программы, в котором располагаются алгоритмы и меню

- test$_l ab01.py -$

Листинг 3.1: Функция нахождения расстояния Левенштейна рекурсивно

```
def LevRecursion(str1, str2, output = False):
    if str1 == '' or str2 == '':
        return abs(len(str1) - len(str2))
    if (str1[-1] == str2[-1]):
        penalty = 0
    else:
        penalty = 1
    return min(LevRecursion(str1, str2[:-1]) + 1,
               LevRecursion(str1[:-1], str2) + 1,
               LevRecursion(str1[:-1], str2[:-1]) + penalty
                 )
```

Листинг 3.2: Функция нахождения расстояния Левенштейна матрично

```
def LevMatr(str1, str2, output = False):
    len_i = len(str1) + 1
    len_j = len(str2) + 1
    mtr = [[i + j for j in range(len_j)] for i in range(len_i)]

    for i in range(1, len_i):
        for j in range(1, len_j):
            if (str1[i-1] == str2[j-1]):
                penalty = 0
            else:
                penalty = 1
            mtr[i][j] = min(mtr[i-1][j] + 1,
                            mtr[i][j-1] + 1,
                            mtr[i-1][j-1] + penalty)
    if output:
        PrintMtr(mtr, str1, str2)
    return(mtr[-1][-1])
```

Листинг 3.3: Функция нахождения расстояния Дамерау-Левенштейна рекурсивно

```
def DamLevRecursion(str1, str2, output = False):
    if str1 == "" or str2 == "":
        return abs(len(str1) - len(str2))
    if str1[-1] == str2[-1]:
        penalty = 0
    else:
        penalty = 1
    result = min(DamLevRecursion(str1, str2[:-1])+1,
                 DamLevRecursion(str1[:-1], str2)+1,
                 DamLevRecursion(str1[:-1], str2[:-1]) + penalty)
    if (len(str1) >= 2 and len(str2) >= 2 and str1[-1] == str2[-2]\
         and str1[-2] == str2[-1]):
```

```
        result = min(result, DamLevRecursion(str1[:-2], str2[:-2]) + 1)
    return result
```

Листинг 3.4: Функция нахождения расстояния Дамерау-Левенштейна матрично

```
def DamLevMtr(str1, str2, output = False):
    len_i = len(str1) + 1
    len_j = len(str2) + 1
    mtr = [[i+j for j in range (len_j)] for i in range (len_i)]

    for i in range(1, len_i):
        for j in range(1, len_j):
            if (str1[i-1] == str2[j-1]):
                penalty = 0
            else:
                penalty = 1
            mtr[i][j] = min(mtr[i-1][j] + 1,
                            mtr[i][j-1] + 1,
                            mtr[i-1][j-1] + penalty)
            if (i > 1 and j > 1) and \
               str1[i-1] == str2[j-2] and str1[i-2] == str2[j-1]:
                mtr[i][j] = min(mtr[i][j], mtr[i-2][j-2]+1)
    if output:
        PrintMtr(mtr, str1, str2)
    return mtr[-1][-1]
```

## 3.3 Тесты

Тестирование было организовано с помощью библиотеки **unittest**. Было создано две вариации тестов:

В первой сравнивались результаты функции с реальным результатом.

Во второй сравнивались реузультаты двух функций(рекурсивной и табличной). При сравнении результатов двух функций использовалась функция get$_s tr$, .

Листинг 3.5: Функция генерации случайной строки

```
def gain_str(str_len = 10):
    letters = string.ascii_lowercase
    return ''.join(random.choice(letters)for i in range(
        str_len))
```

## 3.4 Сравнительный анализ алгоритмов

Потребляемая память матричных реализаций алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна для строк длинами m и n будет отличаться лишь на одну переменную типа int. Учитывая специфику реализации, можно получить формулу вычисления памяти в байтах;

$X_{matr} = 2 * S_{string} + (m + n) * S_{char} + S_{matr} + S_{list} + c * S_i$, где
$2 * S_{string}$ - 2*sizeof(string) = 2*25 байт;
$(m + n) * S_{char}$ - (m+n)*sizeof(char) = (m+n)*1 байт;
$S_{matr}$ - sizeof(int) * (n+1) * (m+1) = 14 * (n+1)*(m+1);
$S_{list}$ - sizeof(list)+sizeof(list)*(m+1) = 36+36*(m+1)
$c * S_i$ - sizeof(int) * колличество переменных (В Левенштейне - 5, В Дамерау-Левенштейне - 6) = 14*5

В Дамерау-Левенштейне количество занимаемой памяти зависит от глубины рекурсии. Глубина рекурсии равна m+n.

$$X_{recur} = \sum_{i=0}^{m+n} (2 * S_{string} + (m + n + 2 - i) * S_{char} + c * S_i)$$

| len | Lev(R) | DamLev(R) | Lev(T) | DamLev(T) |
|---|---|---|---|---|
| 10 | 1270 | 1290 | 2266 | 2280 |
| 50 | 10350 | 10450 | 38506 | 38520 |
| 100 | 30700 | 30900 | 146806 | 146820 |
| 500 | 553500 | 554500 | 3533206 | 3533220 |

Рис. 3.1: Сравнение памяти, потребляемой алгоритмами

# 4 Исследовательская часть

Был проведен замер времени работы каждого из алгоритмов.

| len | Lev(R) | DamLev(R) | Lev(T) | DamLev(T) |
|---|---|---|---|---|
| 3 | 0.041902 | 0.041926 | 0.014004 | 0.016966 |
| 4 | 0.192564 | 0.201459 | 0.027926 | 0.0299215 |
| 5 | 0.9246923 | 1.0276496 | 0.035903 | 0.033949 |
| 6 | 4.9297239 | 7.0609009 | 0.0438637 | 0.10372233 |
| 7 | 30.5806698 | 31.487080 | 0.0518591 | 0.0548467 |

Рекурсивные реализации сравнимы по времени между собой. При увеличении длины строк становится очевидна выигрышность по времени матричного варианта. Уже при длине в 7 символов матричная реализация в 600 раз быстрее.

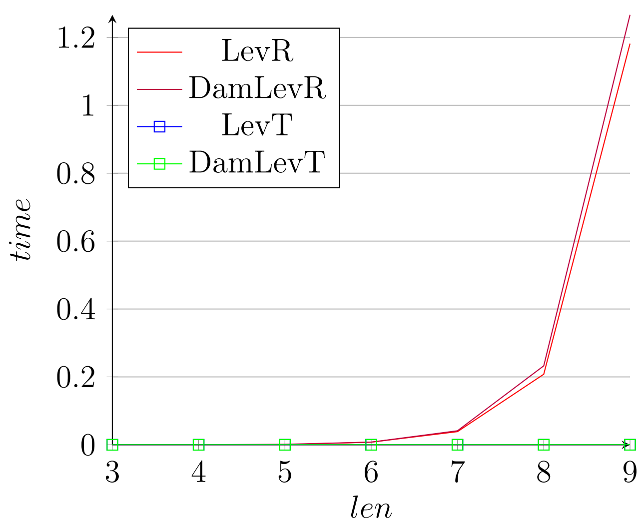


Рис. 4.1: Сравнение времени работы алгоритмов

# Заключение

Был изучен метод динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна. Также изучены алгоритмы Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками, получены практические навыки раелизации указанных алгоритмов в матричной и рекурсивных версиях.

Экспериментально было подтверждено различие во временной эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработаного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк.

В результате исследований я пришел к выводу, что матричная реализация данных алгоритмов заметно выигрывает по времени при росте длины строк.

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

3. R. A. Wagner, M. J. Fischer. The string-to-string correction problem. J. ACM 21 1 (1974). P. 168—173