МГТУ им. Баумана

Лабораторная работа №1

По курсу: "Анализ алгоритмов"

Расстояние Левенштейна

Работу выполнил: Мирзоян Сергей, ИУ7-55Б

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
- 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 временной 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. Буквы верхнего и нижнего регистра считаются разными

Требования к программе:

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

3 Технологическая часть

3.1 Выбор ЯП

В качестве языка программирования был выбран python т.к. я знакомас данным языком, имею представление о способах тестирования программы в рамках данного языка.

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

3.2 Сведения о модулях программы

Программа состоит из:

- analys$_1.py - , test.py -$

Листинг 3.1: Функция нахождения расстояния Левенштейна рекурсивно

```
def levenshtein_rec(a, b):
    if a == "":
        return len(b)
    if b == "":
        return len(a)
    if a[-1] == b[-1]:
        d = 0
    else:
        d = 1
    res = min([levenshtein_rec(a[:-1], b)+1,
       levenshtein_rec(a, b[:-1])+1, levenshtein_rec(a
       [:-1], b[:-1]) + d])
    return res
```

Листинг 3.2: Функция нахождения расстояния Левенштейна матрично

```
def levenshtein_matrix(seq1, seq2):
    size_x = len(seq1) + 1
    size_y = len(seq2) + 1
    matrix = np.zeros((size_x, size_y))
    for x in range(size_x):
        matrix[x, 0] = x
    for y in range(size_y):
        matrix[0, y] = y

    for x in range(1, size_x):
        for y in range(1, size_y):
            if seq1[x-1] == seq2[y-1]:
                matrix[x, y] = min(
                    matrix[x-1, y] + 1,
                    matrix[x-1, y-1],
                    matrix[x, y-1] + 1)
            else:
                matrix[x, y] = min(
                    matrix[x-1, y] + 1,
                    matrix[x-1, y-1] + 1,
                    matrix[x, y-1] + 1)

    return int(matrix[size_x - 1, size_y - 1])
```

Листинг 3.3: Функция нахождения расстояния Дамерау-Левенштейна рекурсивно

```
def D_L_rec(a, b):
    if a == "":
        return len(b)
    if b == "":
        return len(a)
    res = 1
    if len(a) >= 2 and len(b) >= 2:
        if a[len(a)-1] == b[len(b)-2] and a[len(a)-2] == b[
            len(b)-1]:
            if a[-1] == b[-1]:
                d = 0
            else:
```

```
                d = 1
            res = min(
                D_L_rec(a[:-1],b)+1,
                D_L_rec(a[:-1],b[:-1]) + d,
                D_L_rec(a,b[:-1])+1,
                D_L_rec(a[:-2],b[:-2]) + 1)
        else:
            if a[-1] == b[-1]:
                d = 0
            else:
                d = 1
            res = min(
                D_L_rec(a[:-1],b)+1,
                D_L_rec(a[:-1],b[:-1]) + d,
                D_L_rec(a,b[:-1])+1)
    return res
```

Листинг 3.4: Функция нахождения расстояния Дамерау-Левенштейна матрично

```
def D_L_M(seq1, seq2):
    size_x = len(seq1) + 1
    size_y = len(seq2) + 1
    matrix = np.zeros((size_x, size_y))
    for x in range(size_x):
        matrix[x, 0] = x
    for y in range(size_y):
        matrix[0, y] = y
    val = 0
    for x in range(1, size_x):
        for y in range(1, size_y):
            if seq1[x-1] == seq2[y-1]:
                matrix[x, y] = min(
                    matrix[x-1, y] + 1,
                    matrix[x-1, y-1],
                    matrix[x, y-1] + 1)
            else:
                val = 1
                matrix[x, y] = min(
                    matrix[x-1, y] + 1,
                    matrix[x-1, y-1] + 1,
```

```
                    matrix[x, y-1] + 1)
            if x and y and (seq1[x-1] == seq2[y-2] and seq1
                [x-2] == seq2[y-1]):
                matrix[x,y] = min(
                    matrix[x,y],
                    matrix[x-2,y-2]+val)
    return int(matrix[size_x - 1, size_y - 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 RandomString, которая генерирует случайную строку нужной длины.

Листинг 3.5: Функция генерации случайной строки

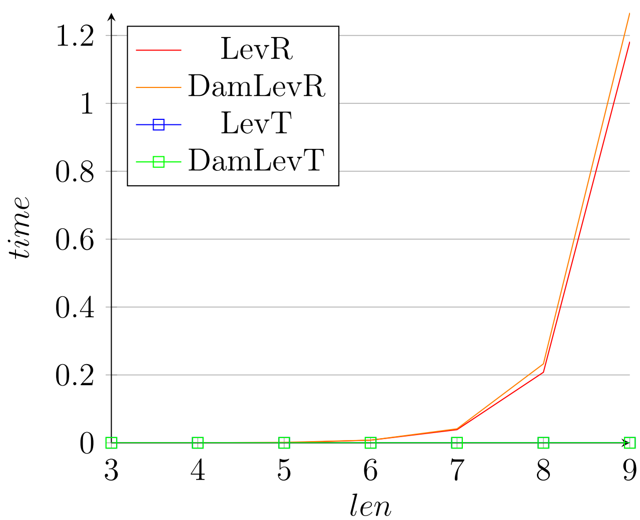
```
def RandomString(strLength = 5):
    letters = string.ascii_lowercase
    return ''.join(random.choice(letters) for i in range(
        strLength))
```

4 Исследовательская часть

Был проведен замер времени работы каждого из алгоритмов.

-

len	Lev(R)	DamLev(R)	Lev(T)	DamLev(T)
3	0.00006	0.00006	0.00003	0.00003
4	0.00033	0.00027	0.00003	0.00003
5	0.00141	0.00143	0.00005	0.00005
6	0.00780	0.00787	0.00005	0.00006
7	0.03876	0.04130	0.00007	0.00007
8	0.20780	0.23259	0.00008	0.00013
9	1.18171	1.26665	0.00009	0.00012



Рекурсивные реализации сравнимы по времени между собой. При увеличении длины строк становится очевидна выигрышность по времени матричного варианта. Уже при длине в 9 символов матричная реализация в 10,000 раз быстрее.

Заключение

Мы получили экспериментальное подтверждение различия 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.