МГТУ им. Баумана

Лабораторная работа №1

По курсу: "Анализ алгоритмов"

Расстояние Левенштейна

Работу выполнила: Оберган Татьяна, 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 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

В качестве языка программирования был выбран python т.к. я знакома с 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:

- main.py - главный файл программы, в котором располагаются алгоритмы и меню
- test.py - файл с тестами

Листинг 3.1: Функция нахождения расстояния Левенштейна рекурсивно

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

Листинг 3.2: Функция нахождения расстояния Левенштейна матрично

```
def LevTable(str1, str2, output = False):
    len_i = len(str1) + 1
    len_j = len(str2) + 1
    table = [[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):
            forfeit = 0 if (str1[i - 1] == str2[j - 1])
               else 1
            table[i][j] = min(table[i - 1][j] + 1,
                              table[i][j - 1] + 1,
                              table[i - 1][j - 1] + forfeit
                                 )
    if output:
        OutputTable(table, str1, str2)
    return(table[-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))
    forfeit = 0 if (str1[-1] == str2[-1]) else 1
    res = min(DamLevRecursion(str1, str2[:-1]) + 1,
              DamLevRecursion(str1[:-1], str2) + 1,
              DamLevRecursion(str1[:-1], str2[:-1]) +
                 forfeit)
    if (len(str1) >= 2 and len(str2) >= 2 and str1[-1] ==
       str2[-2] and str1[-2] == str2[-1]):
        res = min(res, DamLevRecursion(str1[:-2], str2
           [:-2]) + 1)
    return res
```

Листинг 3.4: Функция нахождения расстояния Дамерау-Левенштейна матрично

```
def DamLevTable(str1, str2, output = False):
    len_i = len(str1) + 1
```

```
3       len_j = len(str2) + 1
4       table = [[i + j for j in range(len_j)] for i in range(
            len_i)]
5
6       for i in range(1, len_i):
7           for j in range(1, len_j):
8               forfeit = 0 if (str1[i - 1] == str2[j - 1])
                    else 1
9               table[i][j] = min(table[i - 1][j] + 1,
10                                table[i][j - 1] + 1,
11                                table[i - 1][j - 1] + forfeit
                                      )
12              if (i > 1 and j > 1) and str1[i-1] == str2[j-2]
                    and str1[i-2] == str2[j-1]:
13                  table[i][j] = min(table[i][j], table[i-2][j
                        -2] + 1)
14      if output:
15          OutputTable(table, str1, str2)
16      return(table[-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 RandomString, которая генерирует случайную строку нужной длины.

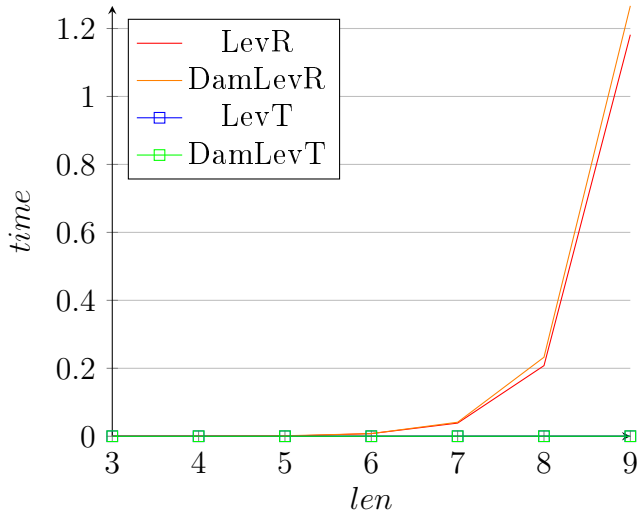
Листинг 3.5: Функция генерации случайной строки

```
1 def RandomString(strLength = 5):
2       letters = string.ascii_lowercase
3       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

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 пришла к выводу, что матричная реализация данных алгоритмов заметно выигрывает по времени при росте длины строк.