Государственное образовательное учреждение высшего профессионального образования
«Московский Государственный Технический Университет имени Н. Э. Баумана»

ОТЧЕТ
По лабораторной работе №1
По курсу «Анализ алгоритмов»
Тема: «Расстояние Левенштейна»

Студент:	Лумбунов Д.В.
Группа:	ИУ7-54
Преподаватель:	

Москва, 2019

Постановка задачи

Расстояние Левенштейна (*редакционное расстояние*, *дистанция редактирования*) — минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

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. Является модификацией расстояния Левенштейна: к операциям вставки, удаления и замены символов, определённых в расстоянии Левенштейна добавлена операция транспозиции (перестановки) символов.

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.

Описание алгоритмов

Рекурсивный алгоритм (Левенштейн)

Здесь и далее считается, что элементы строк нумеруются с первого, как принято в математике, а не с нулевого, как принято во многих языках программирования.

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

$d(S_1, S_2) = D(M,N)$, где

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

$$m(a,b) = \begin{cases} 0, & a = b \\ 1, & a \neq b \end{cases}$$

Рекурсивный алгоритм напрямую реализует эту формулу.

Функция 1 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):

   3.1 Сумма цены преобразования строки a' в b и цены проведения операции удаления, которая необходима для преобразования a в a';

   3.2 Сумма цены преобразования строки a в b' и цены проведения операции вставки, которая необходима для преобразования b' в b;

   3.3 Сумма цены преобразования из a' в b' и операции замены, предполагая, что a и b оканчиваются разные символы;

   3.4 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 формулы 1 может быть малоэффективна при больших i, j, т. к. множество промежуточных значений $D(i, j)$, $1 \leq i \leq |a|$, $1 \leq j \leq |b|$ вычисляются заново множество раз подряд. Для оптимизации нахождения расстояния Левенштейна предложено использовать матрицу (двумерный массив) в целях хранения соответствующих промежуточных значений. В таком случае алгоритм представляет собой построчное заполнение матрицы $A|a|,|b|$ значениями $D(i, j)$ по формуле 1. 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 $\min(|a|, |b|)$. Такой вариант алгоритма называется построчным и именно он реализован в данной работе в качестве нерекурсивного.

Расстояние Дамерау-Левенштейна.

Между двумя строками a, b определяется функцией $d_{a,b}(|a|,|b|)$ как:

$$d_{a,b}(i,j) = \begin{cases} \max(i,j), & \min(i,j) = 0 \\ min\begin{cases} d_{a,b}(i-1,j)+1 \\ d_{a,b}(i,j-1)+1 \\ d_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \\ d_{a,b}(i-2,j-2)+1 \end{cases}, & i,j > 1 \ and \ a_i = b_{j-1} \ and \ a_{i-1} = b_j \\ min\begin{cases} d_{a,b}(i-1,j)+1 \\ d_{a,b}(i,j-1)+1 \\ d_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases}, & otherwise \end{cases}$$

Где $1_{(a_i \neq b_j)}$ – индикаторная функция, равная нулю при $a_i = b_j$ и 1 в противном случае.

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 $3\min(|a|, |b|)$.

Листинги.

Расстояние Левенштейна:

```
def Levenshtein(str1, str2):
   n = len(str1)
   m = len(str2)
   if not min(n,m):
      return max(n,m)
   return min(Levenshtein(str1, str2[:-1]) + 1,
           Levenshtein(str1[:-1], str2) + 1,
           Levenshtein(str1[:-1], str2[:-1]) + (str1[-1] != str2[-1]))
```

Алгоритм Вагнера-Фишера:

```
def Wagner_Fischer(a, b):
    n = len(a)
    m = len(b)
    if n > m:
        a, b = b, a
        n, m = m, n

    current_row = range(n + 1)
    for i in range(1, m + 1):
        previous_row, current_row = current_row, [i] + [0] * n
        for j in range(1, n + 1):
            add, delete, change = previous_row[j] + 1, current_row[j - 1] + 1, previous_row[j - 1]
            if a[j - 1] != b[i - 1]:
                change += 1
            current_row[j] = min(add, delete, change)

    return current_row[n]
```

Расстояние Дамерау-Левенштейна:

```
def Damerau_Levenshtein(s1, s2):
    d = {}
    n = len(s1)
    m = len(s2)
    for i in range(-1,n+1):
        d[(i,-1)] = i+1
    for j in range(-1,m+1):
        d[(-1,j)] = j+1

    for i in range(n):
        for j in range(m):
            cost = s1[i] != s2[j]
            d[(i,j)] = min(
                           d[(i-1,j)] + 1, # delete
                           d[(i,j-1)] + 1, # insert
                           d[(i-1,j-1)] + cost, # substitute
                          )
            if i and j and s1[i]==s2[j-1] and s1[i-1] == s2[j]:
                d[(i,j)] = min (d[(i,j)], d[i-2,j-2] + cost) # transposition

    return d[n-1,m-1]
```

Тесты + сравнения

Enter a: aa
Enter b: aaaa

Levenshtein: 2
Time in ticks: 0.0

Wagner_Fischer: 2
Time in ticks (average by 1000): 0.0007279993057250976

Damerau_Levenshtein: 2
Time in ticks (average by 1000): 0.0005459951400756837
…

Enter a: abcdaabb
Enter b: acbdaaba

Levenshtein: 3
Time in ticks: 6.370006942749023

Wagner_Fischer: 3
Time in ticks (average by 1000): 0.0009100034713745116

Damerau_Levenshtein: 2
Time in ticks (average by 1000): 0.0012739987850189209
…

Enter a: samestr
Enter b: samestr

Levenshtein: 0
Time in ticks: 2.1840022563934327

Wagner_Fischer: 0
Time in ticks (average by 1000): 0.0005460038185119629

Damerau_Levenshtein: 0
Time in ticks (average by 1000): 0.0010919989585876464
…

Сравнение временных хар-к:

Enter a: ti
Enter b: ny

Levenshtein: 2
Time in ticks: 0.0

Wagner_Fischer: 2
Time in ticks (average by 1000): 0.0003640083312988281

Damerau_Levenshtein: 2
Time in ticks (average by 1000): 0.0003639996528625488
…

Levenshtein: 4
Time in ticks: 0.0

Wagner_Fischer: 4
Time in ticks (average by 1000): 0.0008736104488372802

Damerau_Levenshtein: 4
Time in ticks (average by 1000): 0.0004550234317779541
…

Enter a: usual
Enter b: string

Levenshtein: 6
Time in ticks: 0.3639996528625488

Wagner_Fischer: 6
Time in ticks (average by 1000): 0.0007279993057250976

Damerau_Levenshtein: 6
Time in ticks (average by 1000): 0.0007279993057250976
…

Enter a: significant
Enter b: stringtotry

Levenshtein: 10
Time in ticks: 891.1309880256653

Wagner_Fischer: 10
Time in ticks (average by 1000): 0.001456002950668335

Damerau_Levenshtein: 9
Time in ticks (average by 1000): 0.002366002082824707
…

Таблицы измерений времени

Levenshtein				
Strlen2\strlen1	**8**	**6**	**4**	**2**
2	**~0**	**~0**	**~0**	**~0**
4	**~0.0111**	**~0.0074**	**~0.0057**	
6	**~0.1448**	**~0.0334**		
8	**~0.9746**			

Wagner_Fischer				
Strlen2\strlen1	**8**	**6**	**4**	**2**
2	**~0**	**~0**	**~0**	**~0**
4	**~0.0017**	**~0.0008**	**~0.0007**	
6	**~0.0334**	**~0.1448**		
8	**~0.001**			

Выводы

При использовании рекурсивного алгоритма потребление времени при увеличении размера данных растет экспоненциально. При длине строк strlen1=strlen2=8 уже требуется примерно 1 секунда для вычисления результата и это приблизительно в 1000 раз больше, чем требуется при применении алгоритма Вагнера-Фишера.
Применение рекурсивного алгоритма крайне неэффективно по времени, рекомендуется использовать алгоритм Вагнера-Фишера.

Заключение

В результате выполнения данной работы рассмотрены и изучены понятия расстояния Левенштейна и расстояния Дамерау-Левенштейна. Реализованы два варианта алгоритма нахождения расстояния Левенштейна (рекурсивного и нерекурсивного вида). Сравнены их временные характеристики как следствие проведённых экспериментов. Реализован алгоритм нахождения расстояния Дамерау-Левенштейна. Были сделаны выводы о временной эффективности рекурсивного и нерекурсивного алгоритмов.