Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

Лабораторная работа №1

По курсу: "Анализ алгоритмов"

Расстояние Левенштейна

Работу выполнила: Сукочева Алис, ИУ7-53Б

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

Москва, 2020

Оглавление

Введение

В данной лабораторной работе мы познакомимся с расстоянием Левенштейна. Данное расстояние показывает нам минимальное количество редакторских операций (вставки, замены и удаления), которые необходимы нам для перевода одной строки в другую. Это расстояние помогает определить схожесть двух строк.

Какие ошибки человек может допускать при написании какого-то текста? Например, его пальцы могут нажимать на нужные клавиши не в том порядке. С этой проблемой поможет нам справится расстояние Дамерау-Левенштейна. Данное расстояние задействует еще одну редакторскую операцию - транспозицию.

Практическое применение расстояние Левенштейна:

- Сравнение введенной строки со словарными словами в поисковой системе, такой как 'yandex' или 'google'
- Помогает найти разницу двух ДНК. Оценка мутации.

Целью данной работы является разбор и реализация алгоритма Дамерау-Левенштейна и Левенштейна.

В рамках выполнения работы необходимо решить следующие задачи:

1. Изучить алгоритмы Дамерау-Левенштейна и Левенштейна.
2. Реализовать изученные алгоритмы, а также матричную и рекурсивную реализацию алгоритма.
3. Подсчет времени поиска расстояния.
4. Сравнить временные характеристики, а также затраченную память.
5. Описать выбранную среду разработки и ЯП.

1 | Аналитическая часть

1.1 Некоторые теоретические сведения

При преобразовании одного слова в другое мы можем использовать следующие операции:

1. D (*от англ. delete*) - удаление.
2. I (*от англ. insert*) - вставка.
3. R (*от англ. replace*) - замена.

Будем считать стоимость каждой вышеизложенной операции - 1.

Введем понятие совпадения - M (*от англ. match*). Его стоимость будет равна 0.

1.2 Расстояние Левенштейна

Имеем две строки S_1 и S_2, длинной M и N соответственно. Расстояние Левенштейна рассчитывается по приведенной ниже рекуррентной формуле.

!!! По формуле и ссылка на нее

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

1.3 Расстояние Дамерау-Левенштейна

Как было написано выше, в расстоянии Дамерау-Левенштейна задействует еще одну редакторскую операцию - транспозицию T (*от англ. transposition*). Рекуррентная формула Дамерау-Левенштейна представлена ниже.

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

1.4 Вывод

Мы познакомились с основополагающими материалами, которые в дальнейшем помогут нам при реализации алгоритмов Левенштейна и Дамерау-Левенштейна.

2 | Конструкторская часть

В данном разделе мы рассмотрим схемы вышеизложенных алгоритмов.

2.1 Разработка алгоритмов

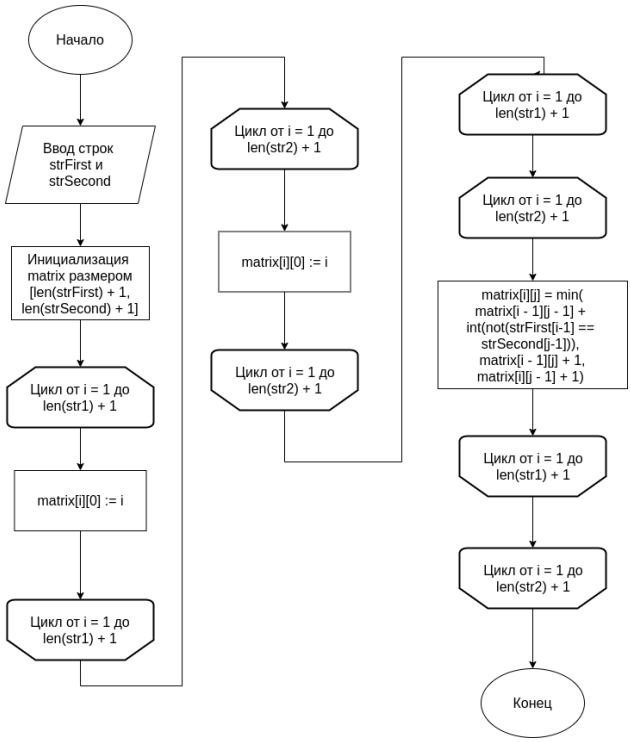

Рис. 2.1: Схема алгоритма Левенштейна

!!! На рисунке таком-то мы разобрали то-то и то-то

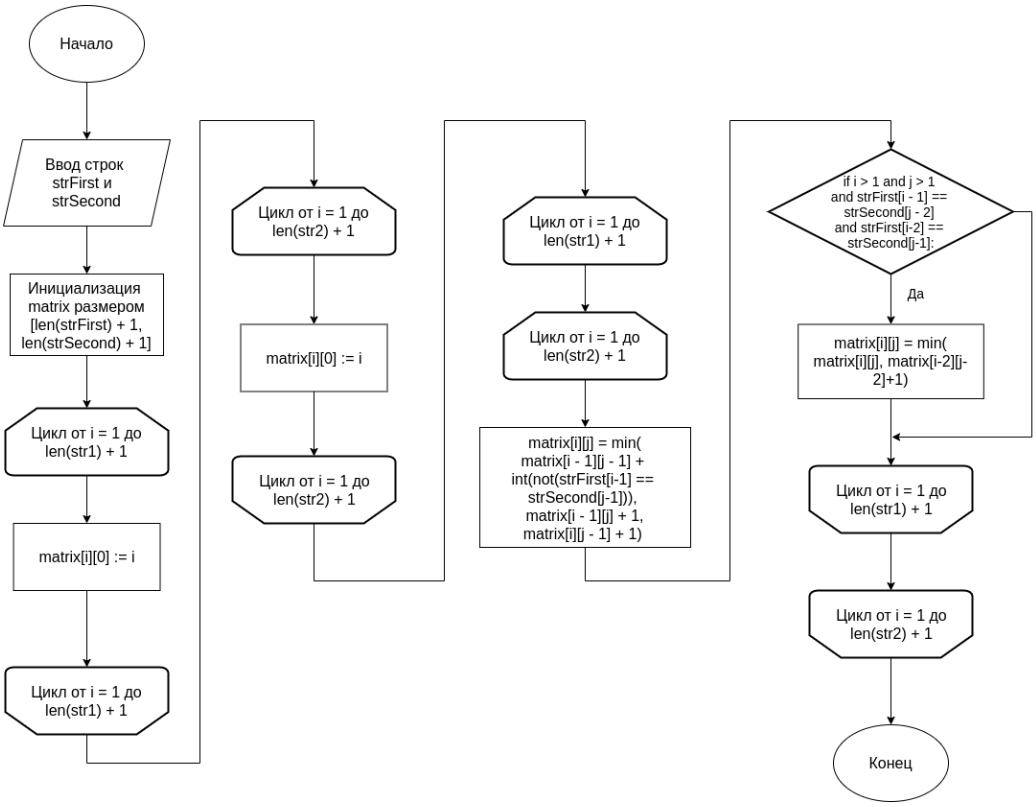

Рис. 2.2: Схема алгоритма Дамерау-Левенштейна

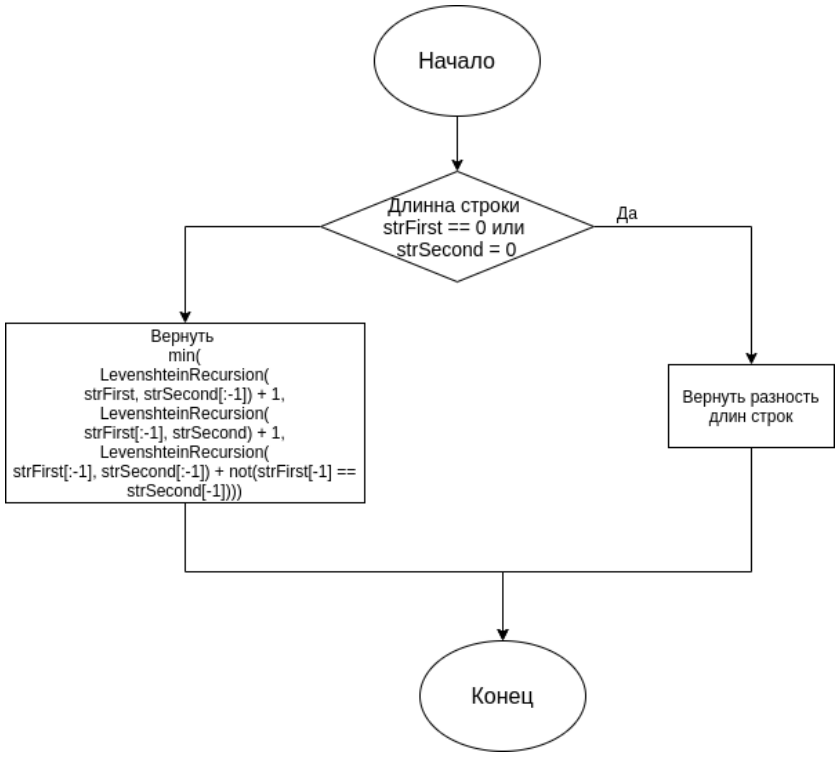

Рис. 2.3: Схема рекурсивного алгоритма Левенштейна

3 | Технологическая часть

3.1 Выбор ЯП

!!! Тутт ссылки на питон и тд...

В данной лабораторной работе использовался язык программирования - python. Данный язык простой и понятный, также я знакома с ним.

Поэтому данный язык был выбран. В качестве среды разработки я использовала Visual Studio Code, т.к. считаю его достаточно удобным и легким. Visual Studio Code подходит не только для Windows, но и для Linux, это еще одна причина, по которой я выбрала VS code, т.к. у меня установлена ОС Ubuntu 18.04.4.

3.2 Требования к программному обеспечению

Входными данными являются две строки. Строки регистрозависимые. На выходе мы получаем матрицу и дистанцию, полученную алгоритмом Левенштейна. Результат алгоритма Дамерау-Левенштейна выводится только в том случае, когда его результат не совпадает с результатом алгоритма Левенштейна. Также требуется замерить время работы каждой реализации.

3.3 Сведения о модулях программы

Данная программа разбита на модули:

- main.py - Файл, содержащий точку входа в программу. В нем происходит общение с пользователем и вызов алгоритмов.

- algorithms.py - Файл содержит непосредственно сами алгоритмы.
- test_time.py - Файл с замером времени работы алгоритмов.

Листинг 3.1: Главная функция main

```
def main():
  output("Enter the first string", GREEN)  # Column
  strFirst = input()
  output("Enter the second string:", GREEN)  # Row
  strSecond = input()
  distanceLev = Levenshtein(strFirst, strSecond, True)
  distanceDamLev = DamerauLevenshtein(strFirst, strSecond)
  output("distance Levenshtein: " + str(distanceLev), GREEN
     )
  if distanceLev != distanceDamLev:
    output("distance Damerau-Levenshtein: " + str(
       distanceDamLev), GREEN)
```

Листинг 3.2: Функция нахождения расстояния Левенштейна матрично

```
def Levenshtein(strFirst, strSecond, flag=False):
  n, m = len(strFirst), len(strSecond)
  matrix = np.full((n + 1, m + 1), 0)  # math.inf)

  matrix[0][0] = 0
  for i in range(1, n + 1):
    matrix[i][0] = i
  for i in range(1, m + 1):
    matrix[0][i] = i

  for i in range(1, n + 1):
    for j in range(1, m + 1):
      matrix[i][j] = min(
      matrix[i - 1][j - 1] +
      int(not(strFirst[i-1] == strSecond[j-1])),  # R
      matrix[i - 1][j] + 1,                       # D
      matrix[i][j - 1] + 1)                       # I

  if flag:
    OutputMatrix(matrix, strFirst, strSecond)
```

```
21
22    return matrix[-1][-1]
```

Листинг 3.3: Рекурсивная функция нахождения расстояния Левенштейна

```
1  def LevenshteinRecursion(strFirst, strSecond):
2    if (strFirst == "" or strSecond == ""):
3      return abs(len(strFirst) - len(strSecond))
4
5    temp = 0 if strFirst[-1] == strSecond[-1] else 1
6    return min(
7    LevenshteinRecursion(strFirst, strSecond[:-1]) + 1, # I
8    LevenshteinRecursion(strFirst[:-1], strSecond) + 1, # D
9    LevenshteinRecursion(strFirst[:-1], strSecond[:-1]) +
       temp # R
10 )
```

Листинг 3.4: Функция нахождения расстояния Дамерау-Левенштейна матрично

```
1  def DamerauLevenshtein(strFirst, strSecond, flag=False):
2    n, m = len(strFirst), len(strSecond)
3    matrix = np.full((n + 1, m + 1), 0)  # math.inf)
4
5    matrix[0][0] = 0
6    for i in range(1, n + 1):
7      matrix[i][0] = i
8    for i in range(1, m + 1):
9      matrix[0][i] = i
10
11   for i in range(1, n + 1):
12     for j in range(1, m + 1):
13       matrix[i][j] = min(
14       matrix[i - 1][j - 1] +
15       int(not(strFirst[i-1] == strSecond[j-1])),  # R
16       matrix[i - 1][j] + 1,                       # D
17       matrix[i][j - 1] + 1)                       # I
18       if i>1 and j>1 and strFirst[i-1] == strSecond[j-2] \
19       and strFirst[i-2] == strSecond[j-1]:
```

```
20          matrix[i][j] = min(matrix[i][j], matrix[i-2][j-2] +
                1) # T
21
22    if flag:
23      OutputMatrix(matrix, strFirst, strSecond)
24    return matrix[-1][-1]
```

Листинг 3.5: Рекурсивная функция нахождения расстояния Дамерау-Левенштейна

```
1  def DamerauLevenshteinRecursion(strFirst, strSecond):
2    if (strFirst == "" or strSecond == ""):
3      return abs(len(strFirst) - len(strSecond))
4
5    temp = 0 if strFirst[-1] == strSecond[-1] else 1
6    result = min(
7    DamerauLevenshteinRecursion(
8    strFirst, strSecond[:-1]) + 1,            # I
9    DamerauLevenshteinRecursion(
10   strFirst[:-1], strSecond) + 1,            # D
11   DamerauLevenshteinRecursion(
12   strFirst[:-1], strSecond[:-1]) + temp     # R
13   )
14
15   if len(strFirst) > 1 and len(strSecond) > 1 and \
16     strFirst[-1] == strSecond[-2] and strFirst[-2] ==
         strSecond[-1]:
17   result = min(result, DamerauLevenshteinRecursion(
18     strFirst[:-2], strSecond[:-2]) + 1)  # T
19
20   return result
```

Листинг 3.6: функция замера времени

```
1  def TimeTest(strFirst, strSecond, countOperations):
2    t1 = time.process_time()
3    for _ in range(countOperations):
4    Levenshtein(strFirst, strSecond)
5    t2 = time.process_time()
6    print("Levenshtein = ", t2 - t1)
7
```

```
    t1 = time.process_time()
    for _ in range(countOperations):
    LevenshteinRecursion(strFirst, strSecond)
    t2 = time.process_time()
    print("Levenshtein (Recursion)= ", t2 - t1)

    t1 = time.process_time()
    for _ in range(countOperations):
    DamerauLevenshtein(strFirst, strSecond)
    t2 = time.process_time()
    print("DamerauLevenshtein = ", t2 - t1)

    t1 = time.process_time()
    for _ in range(countOperations):
    DamerauLevenshteinRecursion(strFirst, strSecond)
    t2 = time.process_time()
    print("DamerauLevenshtein (Recursion)= ", t2 - t1)
```

3.4 Тестирование

Ниже представлены таблицы, в которых четко отражено тестирование программы. Первый и второй столбец отвечают за введенные пользователем слова.

Таблица 3.1: Таблица тестов

Слово 1	Слово 2	Ожидаемый вывод	Вывод программы
сито	столб	3	3
exponential	polynomial	6	6
Alice	Alice	0	0
Alice	alice	1	1
ma	am	2 1	2 1
		0	0
abc	cab	2	2

Все тесты пройдены успешно.

3.5 Вывод

Мы рассмотрели листинги кода, обосновали выбор использованного в данной работе языка программирования и среды разработки, а также убедились в корректной работе программы

Заключение

В этой лабораторной работе мы познакомились с алгоритмами Левенштейна и Дамерау-Левенштейна. Построили схемы, соответствующие данным алгоритмам, также разобрали рекуррентные реализации. Написали полностью готовый и протестированный программный продукт, который считает дистанцию 4 способами.