МГТУ им. Баумана

Лабораторная работа №1

По курсу: "Анализ алгоритмов"

Расстояние Левенштейна

Работу выполнил: студент группы ИУ7-53Б
Наместник Анастасия

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

Москва, 2020

Оглавление

Введение

Расстояние Левенштейна (редакционное расстояние) - минимальное количество односимвольных операций (а именно вставки, удаления, замены), необходимых для превращения одной последовательности символов в другую.

Редакционным предписанием называется последовательность действий, необходимых для получения из первой строки второй кратчайшим образом.

Расстояние Левенштейна на практике используется для решения таких задач, как:

- исправление ошибок в слове (в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи);
- сравнение текстовых файлов утилитой diff и ей подобными. Здесь роль «символов» играют строки, а роль «строк» — файлы;
- сравнение генов, хромосом и белков в биоинформатике.

Целью данной лабораторной работы является изучение, реализация и сравнительный анализ алгоритмов Левенштейна и Дамерау-Левенштейна.

В данной лабораторной работе требуется решить четыре задачи.

1. Изучить алгоритмы Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками.
2. Получить практические навыки реализации указанных алгоритмов: двух алгоритмов в матричной версии и двух алгоритмов в рекурсивной версии.

3. Сделать сравнительный анализ по затрачиваемым памяти и времени.

4. Описать и обосновать полученные результаты в отчете о выполненной лабораторно работе.

1 Аналитическая часть

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

Обозначение операций, применимых к символам:

1. D (англ. delete) — удаление;
2. I (англ. insert) — вставка;
3. R (replace) — замена;
4. M (match) - совпадение.

1.1 Расстояние Левенштейна

Пусть S_1 и S_2 — две строки (длиной M и N соответственно) над некоторым алфавитом, тогда расстояние Левенштейна можно подсчитать по рекуррентной формуле 1.1:

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ min(& \\ D(i,j-1) + 1, & \\ D(i-1,j) + 1, & j > 0, i > 0 \\ D(i-1,j-1) + m(S_1[i], S_2[j]) & \\ ), & \end{cases} \tag{1.1}$$

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

1.2 Расстояние Дамерау-Левенштейна

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

Расстояние Дамерау-Левенштейна вычисляется по рекуррентной формуле 1.2:

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ min(& \\ D(i, j-1) + 1, & \\ D(i-1, j) + 1, & j > 0, i > 0 \\ D(i-1, j-1) + m(S_1[i], S_2[j]) & \\ \left[\begin{array}{l} D(i-2, j-2) + 1, \text{ если } i,j > 1 \text{ и } a_i = b_{j-1}, a_{i-1} = b_j, \\ \qquad \infty \text{ , иначе)} \end{array} \right. & \end{cases} \tag{1.2}$$

Каждый рекурсивный вызов соответствует одному из случаев:

- $d_{a,b}$(i - 1, j) + 1 соответствует удалению символа (из a в b);
- $d_{a,b}$(i, j - 1) + 1 соответствует вставке (из a в b);
- $d_{a,b}$(i - 1, j - 1) + 1 соответствие или несоответствие, в зависимости от совпадения символов;
- $d_{a,b}$(i - 2, j - 2) + 1 в случае перестановки двух последовательных символов, +inf иначе.

1.3 Вывод

В аналитической части были получены краткие теоретические сведения о поиске расстояния Левенштейна и об аналогичном алгоритме, отличающемся на одну операцию, - алгоритм поиска расстояния Дамерау-

Левенштейна, а также были приведены рекуррентные формулы обоих алгоритмов.

2 | Конструкторская часть

Ниже представлены схемы рассматриваемых алгоритмов.

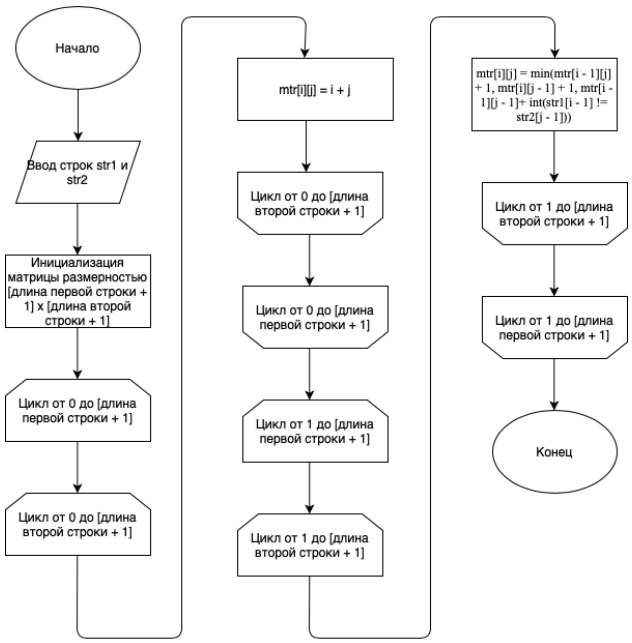

Рис. 2.1: Схема алгоритма Левенштейна (матрица)

2.1 Вывод

В данном разделе были рассмотрены схемы четырех алгоритмов: алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна с матричной и рекурсивной реализацией. Из полученных данных можно ви-

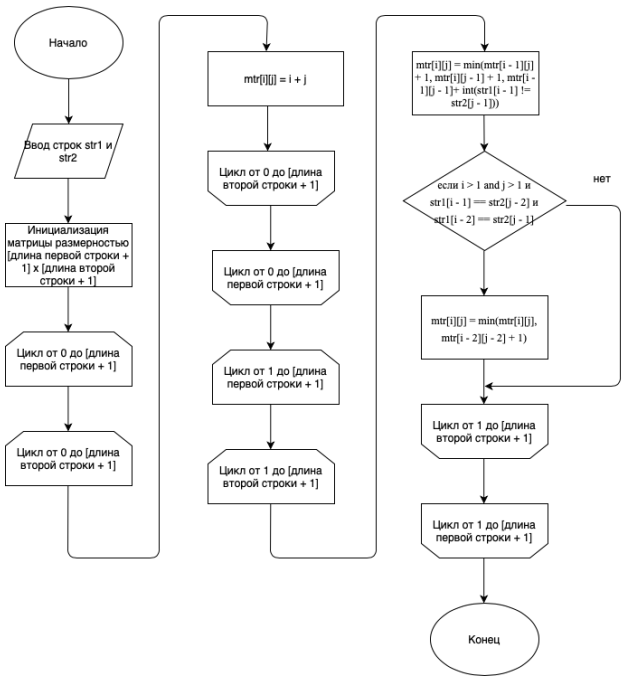

Рис. 2.2: Схема алгоритма Дамерау-Левенштейна (матрица)

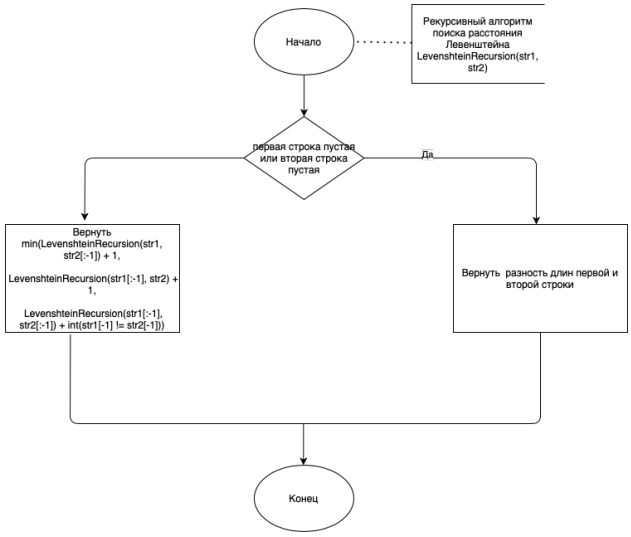

Рис. 2.3: Схема алгоритма Левенштейна (рекурсия)

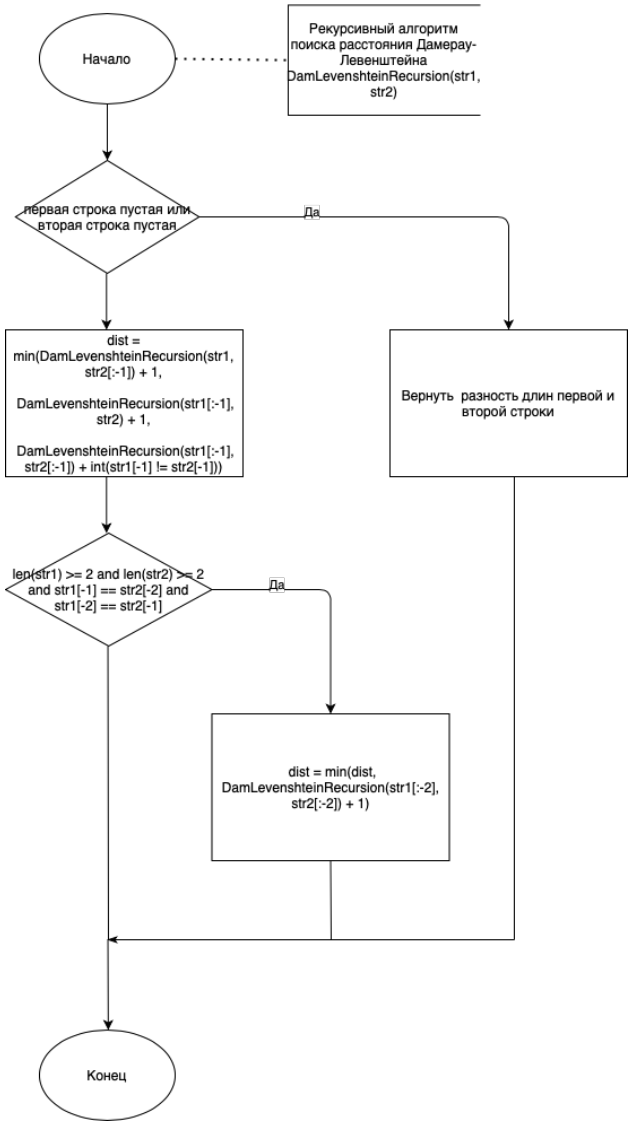

Рис. 2.4: Схема алгоритма Дамерау-Левенштейна (рекурсия)

зуально оценить емкость каждого из алгоритмов и предположить, что рекурсивный подход к расстоянию Левенштейна наименее затратный, с точки зрения написания кода.

3 | Технологическая часть

3.1 Выбор ЯП

В данной лабораторной работе использовался язык программирования - python (3.8.3) [1] в целях упрощения работы со структурами и визуализацией данных сравнительных анализов, а также из-за наличия опыта работы с данным ЯП. В качестве интегрированной среды разработки использовался интерпретатор python - IDLE [2]. Для замеров процедурного времени использовалась функция process_time() библиотеки time [3]. В качестве инструмента измерения затраченной памяти использовалась библиотека memory_profiler [4].

3.2 Сведения о модулях программы

Программа состоит из следующих модулей:

- editorial_distance.py - главный файл программы;
- test.py - файл с тестами;
- time.py - файл с замерами временнных характеристик.

Листинг 3.1: Подпрограмма поиска расстояния Левенштейна с помощью таблицы

```
1 def LevenshteinMatrix(str1, str2):
2     n = len(str1) + 1
3     m = len(str2) + 1
4     mtr = [[(i + j) for j in range(m)] for i in range(n)]
5
```

```
6       for i in range(1, n):
7           for j in range(1, m):
8               mtr[i][j] = min(mtr[i - 1][j] + 1, mtr[i][j -
                    1] + 1,
9                               mtr[i - 1][j - 1] + int(str1[i
                                    - 1] != str2[j - 1]))
10      return mtr
```

Листинг 3.2: Подпрограмма поиска расстояния Левенштейна рекурентно

```
1 def LevenshteinRecursion(str1, str2):
2       if not str1:
3           return len(str2)
4       if not str2:
5           return len(str1)
6       return min(LevenshteinRecursion(str1, str2[:-1]) + 1,
7                  LevenshteinRecursion(str1[:-1], str2) + 1,
8                  LevenshteinRecursion(str1[:-1], str2[:-1]) +
                       int(str1[-1] != str2[-1]))
```

Листинг 3.3: Подпрограмма поиска расстояния Дамерау-Левенштейна с помощью таблицы

```
1 def DamLevenshteinMatrix(str1, str2):
2       n = len(str1) + 1
3       m = len(str2) + 1
4       mtr = [[(i + j) for j in range(m)] for i in range(n)]
5
6       for i in range(1, n):
7           for j in range(1, m):
8               mtr[i][j] = min(mtr[i - 1][j] + 1, mtr[i][j -
                    1] + 1,
9                               mtr[i - 1][j - 1] + int(str1[i
                                    - 1] != str2[j - 1]))
10              if i > 1 and j > 1 and str1[i - 1] == str2[j -
                    2] and str1[i - 2] == str2[j - 1]:
11                  mtr[i][j] = min(mtr[i][j], mtr[i - 2][j -
                        2] + 1)
12
13      return mtr
```

Листинг 3.4: Подпрограмма поиска расстояния Дамерау-Левенштейна рекурентно

```
def DamLevenshteinRecursion(str1, str2):
    if not str1:
        return len(str2)
    if not str2:
        return len(str1)

    dist = min(DamLevenshteinRecursion(str1, str2[:-1]) +
       1,
               DamLevenshteinRecursion(str1[:-1], str2) +
                  1,
               DamLevenshteinRecursion(str1[:-1], str2
                  [:-1]) + int(str1[-1] != str2[-1]))
    if (len(str1) >= 2 and len(str2) >= 2 and str1[-1] ==
       str2[-2] and str1[-2] == str2[-1]):
        dist = min(dist, DamLevenshteinRecursion(str1[:-2],
            str2[:-2]) + 1)
    return dist
```

3.3 Тесты

Было реализовано несколько видов тестов:

- сравнение результата матричного и рекурсивного алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна при заданном количестве тестов и количестве символов в строке, которая генерируется с помощью отдельной процедуры;
- проверка работы обоих алгоритмов с пустой строкой;
- проверка работы обоих алгоритмов, если строки равны.

Листинг 3.5: Функция генерации строки заданной длины

```
def StringGen(strlen):
    l = string.ascii_lowercase
    return ''.join(random.choice(l) for i in range(strlen))
```

Листинг 3.6: Функция проверки работы алгоритмов при пустой строке

```
def EmptyString(func):
    strlen = int(input("Enter the string length: "))
    s = StringGen(strlen)

    if func == LevenshteinMatrix or func ==
       DamLevenshteinMatrix:
        mtr = func(s, "")
        return int(mtr[len(mtr) - 1][len(mtr[len(mtr) - 1])
            - 1] == len(s))
    dist = func(s, "")
    return int(dist == len(s))
```

Листинг 3.7: Функция проверки работы алгоритмов при одинаковых строках

```
def EqualStrings(func):
    strlen = int(input("Enter the string length: "))
    s = StringGen(strlen)

    if func == LevenshteinMatrix or func ==
       DamLevenshteinMatrix:
        mtr = func(s, s)
        return int(mtr[len(mtr) - 1][len(mtr[len(mtr) - 1])
            - 1] == 0)
    dist = func(s, s)
    return int(dist == 0)
```

Листинг 3.8: Функция проверки работы матричного и рекурсивного алгоритма поиска расстояния Левенштейна

```
def cmp_Lev_matrix_recursion():

    print("\033[33m{:<}\n".format(
        "*****Levenshtein distance (matrix/recursion)*****"
           ))
    print("\033[0m")

    t = int(input("Enter the number of tests: "))

    for i in range(t):
```

```
10          count = 0
11
12          strlen = int(input("Enter the string length: "))
13          str1, str2 = StringGen(strlen), StringGen(strlen)
14          mtr, dist = LevenshteinMatrix(
15              str1, str2), LevenshteinRecursion(str1, str2)
16          count += int(mtr[len(mtr) - 1][len(mtr[len(mtr) -
                1]) - 1] == dist)
17          PrintRes(count)
```

Листинг 3.9: Функция проверки работы матричного и рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна

```
1  def cmp_DamLev_matrix_recursion():
2
3      print("\033[33m{:<}\n".format(
4          "*****Damerau-Levenshtein distance (matrix/
               recursion)*****"))
5      print("\033[0m")
6
7      t = int(input("Enter the number of tests: "))
8
9      for i in range(t):
10         count = 0
11
12         strlen = int(input("Enter the string length: "))
13         str1, str2 = StringGen(strlen), StringGen(strlen)
14         mtr, dist = DamLevenshteinMatrix(
15             str1, str2), DamLevenshteinRecursion(str1, str2
                   )
16         count += int(mtr[len(mtr) - 1][len(mtr[len(mtr) -
               1]) - 1] == dist)
17         PrintRes(count)
```

3.4 Вывод

В технологической части были представлены модули программы, листинги кода и тестов к программе, а также обусловлен выбор языка про-

граммирования и приведены использовавшиеся в ходе работы инструменты.

4 | Исследовательская часть

4.1 Результат замеров времени выполнения всех алгоритмов

Сравним временные показатели работы каждого из рассматриваемых четырех алгоритмов при длине строки от 2 до 9 символов. Для этого установим количество итераций (повторений вызова процедуры) iter и воспользуемся библиотекой time для замера времени выполнения каждого алгоритма по iter раз. Возьмем iter = 50, а длины строк равными [2, 3, 4, 5, 6, 7, 8, 9]. Время выполнения алгоритмов со строками небольшой длины довольно мало, поэтому чтобы оценить его, следует взять среднее арифметическое от времени, за которое процедура отработает установленные iter раз. Результат приведен в таблице 4.1:

Таблица 4.1: Сравнительная таблица времени выполнения всех алгоритмов

Длина строки	Лев(м)	Лев(р)	Дам-Лев(м)	Дам-Лев(р)
2	0.0000727	0.0000723	0.0000738	0.0000803
3	0.0000120	0.0000340	0.0000129	0.0000377
4	0.0000190	0.0001767	0.0000203	0.0001908
5	0.0000266	0.0009067	0.0000297	0.0010294
6	0.0000368	0.0049724	0.0000449	0.0057501
7	0.0000499	0.0312771	0.0000601	0.0304088
8	0.0000603	0.1437498	0.0000685	0.1575722
9	0.0000753	0.7829850	0.0000848	0.8673832

Из рисунка 4.1 следует, что рекурсивные алгоритмы Левенштейна и Дамерау-Левенштейна значительно уступают по времени выполнения

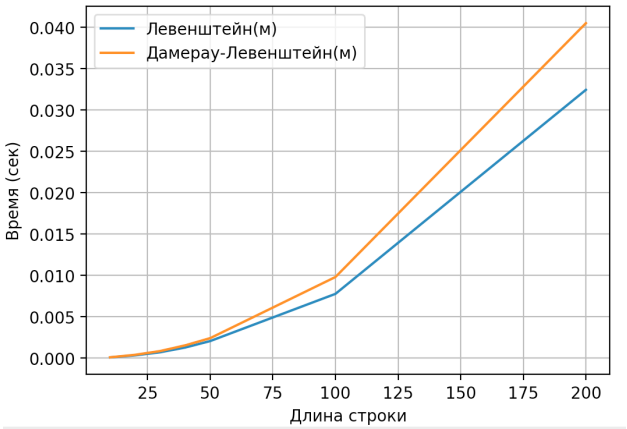

Рис. 4.1: Сравнение времени выполнения всех алгоритмов

соответствующим матричным алгоритмам, причем уже при длине строки в 9 символов рекурсивный алгоритм Дамерау-Левенштейна начинает работать медленнее алгоритма Левенштейна, что обусловливается дополнительной операцией транспозиции. Однако, можно заметить, что при таких длинах строк затруднительно сделать сравнительный анализ матричных алгоритмов, для этого потребуются более длинные строки и большее количество повторений процедур.

4.2 Результат замеров времени выполнения матричных алгоритмов

Для того чтобы получить примерное время работы матричных алгоритмов Левенштейна и Дамерау-Левенштейна, аналогично предыдущему эксперименту, возбмем фиксированные длины строк [10, 20, 30, 40, 50, 100, 200] и количество итераций iter = 500. Суммарное время выполнения процедуры, запущенной iter раз, нужно разделить на iter. Результат представлен в сравнительной таблице 4.2.

Анализ рисунка 4.2, отражающего зависимость времени выполнения матричных алгоритмов Левенштейна и Дамерау-Левенштейна от длины строки, показывает, что при относительно малых длинах строк различия во времени незначительны, однако при длине 100 символов и больше ал-

Таблица 4.2: Сравнительная таблица времени выполнения алгоритмов Левенштейна и Дамерау-Левенштейна в матричной реализации

String length	Левенштейн	Дамерау-Левенштейн
10	0.00009	0.00010
20	0.00034	0.00040
30	0.00071	0.00086
40	0.00128	0.00155
50	0.00206	0.00242
100	0.00777	0.00980
200	0.03242	0.04048

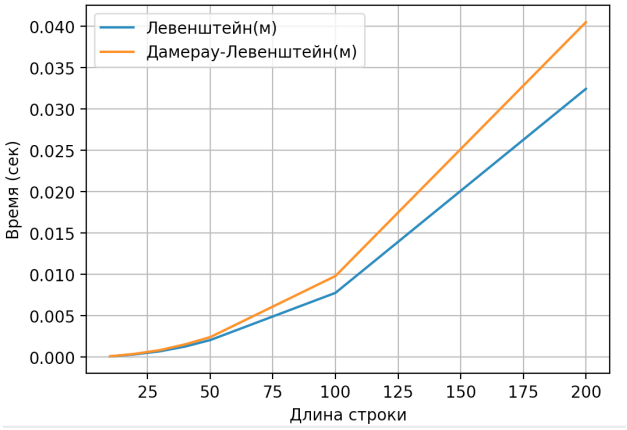

Рис. 4.2: Сравнение времени выполнения алгоритмов Левенштейна и Дамерау-Левенштейна в матричной реализации

горитм Дамерау-Левенштейна работает уже заметно медленнее, и при 200 символах его скорость становится ниже почти в 1.3 раза. Как уже было сказано, такое расхождение во времени объясняется четвертой операцией (транспозиция), утяжеляющей данный алгоритм.

4.3 Анализ затрачиваемой памяти

Память, затрачиваемая на реализацию матричный алгоритмов Левенштейна и Дамерау-Левенштейна, представляет собой произведение размерности матрицы и количество байт, занимаемых типом integer (в языке программирования python - 28 байт). Рекурсивные алгоритмы избыточны за счет повторных вызовов, уже при длине строки в 2 символа количество вызовов рекурсии составит 18 раз. Каждый вызов занимает 32 Мегабайта. Таким образом, рекурсивный подход займет 32x18 = 576 Мегабайт.

4.4 Вывод

Из полученных экспериментальным путем данных очевидно считать алгоритм поиска расстояния Левенштейна, реализованный с помощью матрицы, наиболее оптимальным как с точки зрения временных характеристик, так и относительно затрачиваемой памяти. В то же время результаты сравнительных анализов показывают, что рекуррентный алгоритм поиска расстояния Дамерау-Левенштейна является самым медленным и самым емким по занимаемым ресурсам памяти.

Заключение

В ходе лабораторной работы были реализованы и проанализированы алгоритмы поиска редакционного расстояния Левенштейна и Дамерау-Левенштейна, изучены особенности и отличия данных алгоритмов, а также способы их реализации: матричный, рекурсивный. В целях оценки эффективности алгоритмов, был проведён сравнительный анализ временных характеристик с учетом строк малых и больших длин, который позволил установить, что алгоритмы, реализованные с помощью рекуррентных формул, работают значительно медленнее матричных из-за многочисленных вызовов процедуры с повторяющимися параметрами, а также то, что в самих рекурсивных алгоритмах расхождение во времени заметно уже к длине слова в 9 символов (формула Дамерау-Левенштейна содержит дополнительную операцию). При сравнении матричных алгоритмов было выяснено, что скорость работы алгоритма Дамерау-Левенштейна становится ниже при длине слова в 100 символов, где алгоритм Левенштейна почти в 1.3 раза быстрее. Таким образом, алгоритм Дамерау-Левенштейна и при матричной, и при рекурсивной реализации работает медленнее алгоритма Левенштейна за счёт четвёртой операции - транспозиции. Также можно выделить недостаток рекурсивной реализации у обоих алгоритмов - наличие повторных вызовов, что снижает эффективность такого подхода. Из вышеизложенного справедливо утверждать, что матричный алгоритм поиска расстояния Левенштейна является самым выгодным по затрачиваемым на его выполнение ресурсам, а рекурентный алгоритм поиска расстояния Дамерау-Левенштейна, наоборот, самый медленный и емкий из всех рассмотренных алгоритмов.

Литература

[1] *Майкл, Доусон.* Python Programming for the Absolute Beginner, 3rd Edition / Доусон Майкл. — Прогресс книга, 2019. — P. 416.

[2] *Lutz, M.* The IDLE User Interface / M Lutz. — O'Reilly Media, 2013.

[3] Time. https://docs.python.org/3/library/time.html.

[4] Memory Profiler. https://pypi.org/project/memory-profiler/.