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)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Отчет по лабораторной работе № 1 по курсу “Анализ алгоритмов”

Тема Расстояния Левенштейна и Дамерау-Левенштейна

Студент Ву Хай Данг

Группа ИУ7и-52Б

Оценка (баллы)

Преподаватель Волкова Л. Л.

Москва — 2023 г.

Содержание

Введение

Расстояние Левенштейна и расстояние Дамерау — Левенштейна, представляют собой метрики, используемые в информатике и обработке текстов для измерения различий между двумя строками или последовательностями символов.

Расстояние Левенштейна, также известное как редакционное расстояние или расстояние редактирования, измеряет минимальное количество операций (вставки, удаления и замены символов), необходимых для преобразования одной строки в другую. Эта метрика была названа в честь советского математика Владимира Левенштейна и широко используется в автоматической обработке текста и компьютерной лингвистике.

Расстояние Дамерау — Левенштейна является расширением расстояния Левенштейна, которое также учитывает операцию транспозиции (перестановки соседних символов) как допустимую операцию. Оно получило свое имя в честь американского ученого Фредерика Дамерау, который также внес вклад в развитие этой метрики.

Метод динамического программирования — это способ решения сложных задач путём разбиения из на более простые подзадачи.

Целью данной лабораторной работы является изучение метода динамического программирования на материале алгоритмов Левенштейна и Дамерау—Левенштейна. Задачи данной лабораторной работы следующие:

— описание расстояний Левенштейна и Дамерау — Левенштейна между строками;

— разработка алгоритмов нахождения расстояний Левенштейна и Дамерау — Левенштейна между строками;

— создание программного обеспечения, реализующего перечисленные выше алгоритмы;

— проведение сравнительного анализа реализаций алгоритмов по затраченному процессорному времени и памяти;

— описание и обоснование полученных результатов в отчете о выполненной лабораторной работе.

1 Аналитическая часть

1.1 Расстояние Левенштейна

Расстояние Левенштейна — минимальное количество редакционных операций, необходимых для преобразования одной строки в другую.

Пусть S_1 и S_2 — две строки, длиной N и M соответственно, а редакционные операции:

— вставка символа в произвольной позиции (I — Insert);

— удаление символа в произвольной позиции (D — Delete);

— замена символа на другой (R — Replace).

Принято, что для этих операций “штраф” равен 1.

Для поиска расстояния Левенштейна используют рекуррентную формулу, то есть такую, которая использует предыдущие члены ряда или последовательности для вычисления последующих:

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

$$m(a[i],b[j]) = \begin{cases} 0, & \text{если } S_i = S_j, \\ 1, & \text{иначе} \end{cases} \tag{1.2}$$

Итерационный алгоритм поиска расстояния Левенштейна будет выполнять расчёт по формуле (1.1).

1.2 Расстояние Дамерау – Левенштейна

Расстояние Дамерау — Левенштейна измеряет минимальное количество операций (вставок, удалений, замен и перестановок символов), необходимых для преобразования одной строки в другую. Эта метрика часто используется в информационной теории, лингвистике и компьютерной науке для сравнения и анализа текстовых данных.

По сравнению с расстоянием Левенштейна, расстояние Дамерау — Левенштейна учитывает возможность перестановки символов, что может быть полезным, например, при исправлении опечаток, где буквы могут быть перепутаны местами в словах. Это позволяет более точно измерить сходство между двумя строками и более эффективно обрабатывать различные виды текстовых данных.

Тогда для данного расстояния определены следующие редакционные операции:

— вставка символа в произвольной позиции (I - Insert);

— удаление символа в произвольной позиции (D - Delete);

— замена символа на другой (R - Replace);

— транспозиция двух символов (S - Swap).

Таким образом, чтобы получить формулу нахождения расстояния Дамерау — Левенштейна, необходимо в формулу для поиска расстояния Левенштейна добавить еще одно определение минимума, но уже из четырех вариантов, если возможна перестановка двух соседних символов:

$$D(i,j) = \begin{cases} max(i,j), & min(i,j) = 0 \\ \min\{ & \\ \quad D(i,j-1)+1, & \text{i} > 1,\ \text{j} > 1, \\ \quad D(i-1,j)+1, & S_i = S_{j-1}, \\ \quad D(i-2,j-2)+1, & S_{i-1} = S_j, \\ \quad D(i-1,j-1) + \begin{cases} 0, & \text{если } S_i = S_j, \\ 1, & \text{иначе} \end{cases} & \\ \}, & \\ \min\{ & \\ \quad D(i,j-1)+1, & \\ \quad D(i-1,j)+1, & \text{иначе} \\ \quad D(i-1,j-1) + \begin{cases} 0, & \text{если } S_i = S_j, \\ 1, & \text{иначе} \end{cases} & \\ \} & \end{cases} \tag{1.3}$$

1.2.1 Итерационный алгоритм поиска расстояния Дамерау — Левенштейна

Суть итерационного алгоритма поиска расстояния Дамерау — Левенштейна заключается в построчном заполнении матрицы промежуточных значений расстояния. Результатом является последний элемент последней строки матрицы. При больших строках приходится хранить большие матрицы. Поскольку нам не важны промежуточные значения, можно хранить только текущую и предыдущую строки. Итерационный алгоритм поиска расстояния Дамерау — Левенштейна будет выполнять расчёт по формуле (1.3).

1.2.2 Рекурсивный алгоритм поиска расстояния Дамерау — Левенштейна

Данный алгоритм использует для решения формулу (1.3) и является рекурсивным, а значит, для хранения промежуточных результатов используется стек. Кроме того, при этом подходе возникает проблема повторных вычислений, так как функция $D(s1[1..i], s2[1..j])$ будет выполняться несколько раз в разных ветвях дерева для одних и тех же значений i и j.

1.2.3 Рекурсивный алгоритм поиска расстояния Дамерау — Левенштейна с использованием кеша

В этом алгоритме используется кеш. Кеш позволяет сохранять результаты промежуточных вычислений и избегать повторных вычислений, что значительно повышает эффективность алгоритма по сравнению с рекурсивным алгоритмом без кеша. В качестве кеша используется матрица, её ячейки инициализируются значением -1. Хотя и этот алгоритм, и итерационный алгоритм используют матрицы, они различаются по способу реализации. В этом алгоритме они работают в противоположном направлении.

Вывод

В данном разделе были описаны алгоритмы поиска расстояний Левенштейна и Дамерау — Левенштейна. Эти алгоритмы позволяют найти редакционные расстояния для двух строк.

2 Конструкторская часть

В данном разделе будут представлены требования к программе, схемы алгоритмов поиска расстояний Левенштейна и Дамерау — Левенштейна, выбранные типы данных.

2.1 Требования к программе

К программе предъявляются следующие требования.

— Программа должна предоставлять 2 режима работы: режим расчёта расстояний между введёнными пользователем двумя строками и режим массированного замера процессорного времени выполнения поиска редакционных расстояний реализациями алгоритмов.

— В начале работы программы пользователю нужно ввести целое число — это выбор пункта меню.

К первому режиму работы программы предъявляются следующие требования:

— если пункт меню — число от 1 до 4, то вычислить расстояние, для этого надо ввести две строки;

— две пустые строки — корректный ввод, программа не должна аварийно завершаться;

— программа должна вывести расстояние и, если расстояние вычисляется не рекурсивно, также матрицу D.

Ко второму режиму работы программы предъявляются следующие требования:

— если пункт меню — число 5, то провести замеры времени поиска каждого расстояния реализацией каждого алгоритма;

— длины строк во втором режиме принимаются равными;

— строки заданной длины генерируются автоматически.

2.2 Алгоритмы поиска редакционных расстояний

Схемы алгоритмов поиска редакционных расстояний представлены на рисунках 2.1–2.4. Перед началом расчёта по рекурсивному алгоритму поиска расстояния Дамерау — Левенштейна с кешем требуется инициализировать ячейки матрицы значениями -1.

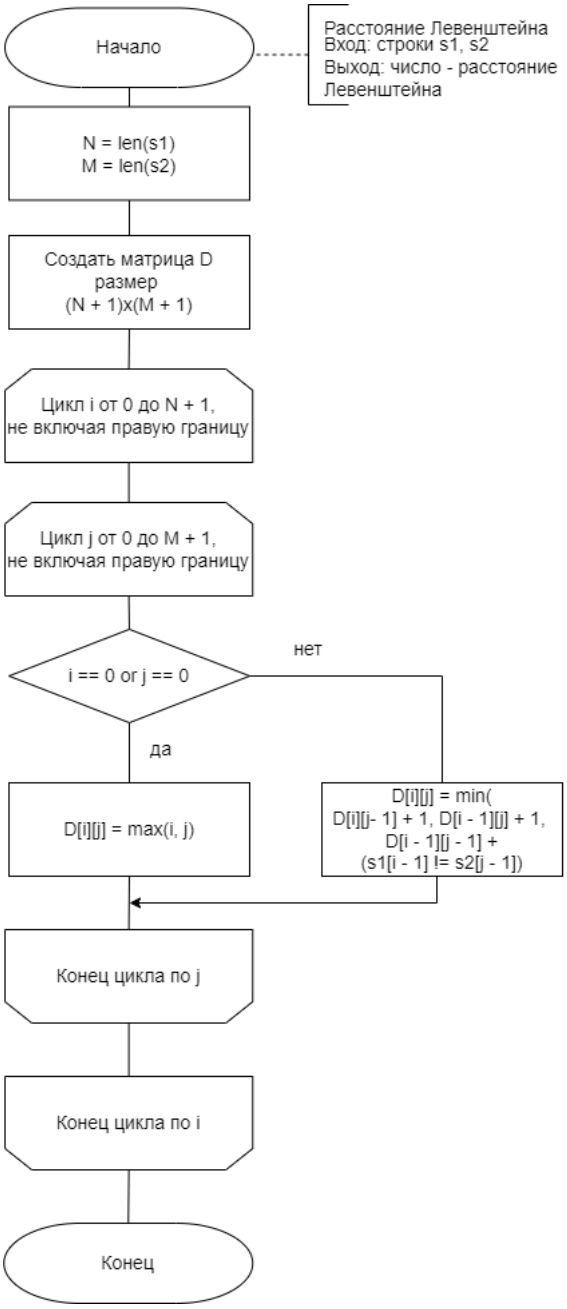

Рисунок 2.1 – Схема итерационного алгоритма поиска расстояния Левенштейна

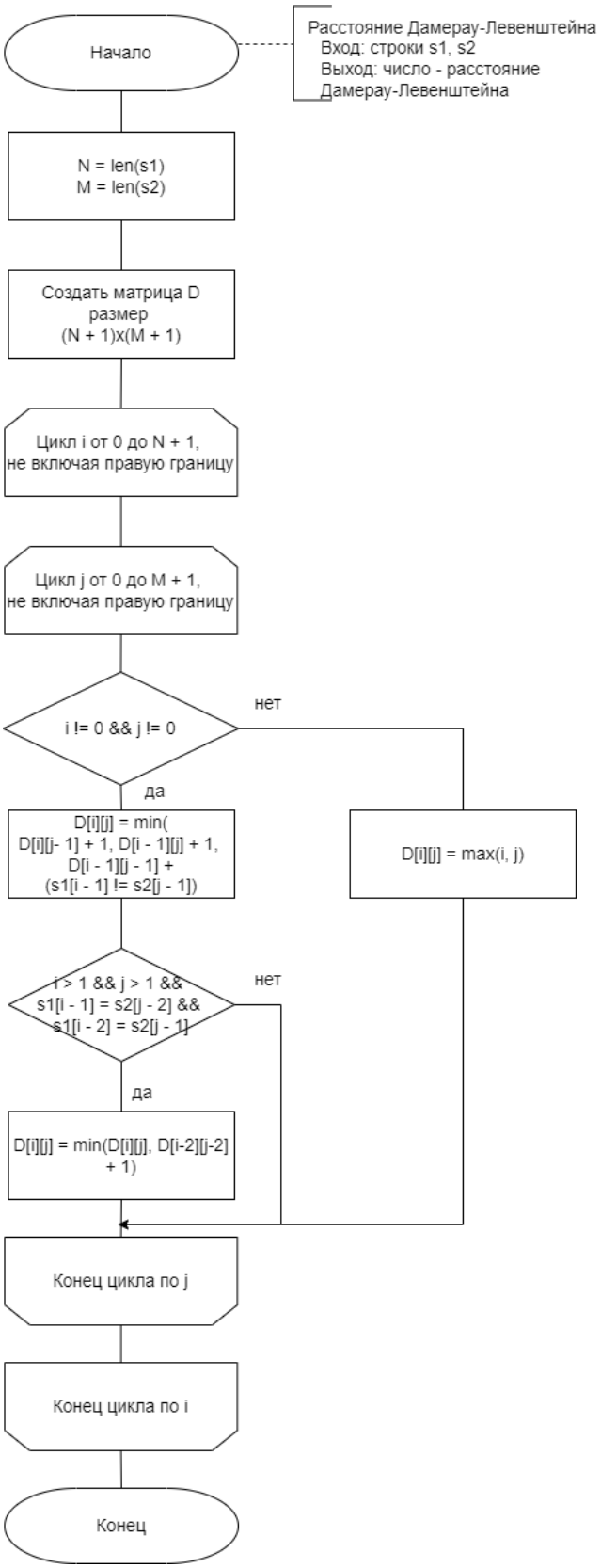

Рисунок 2.2 – Схема итерационного алгоритма поиска расстояния Дамерау — Левенштейна

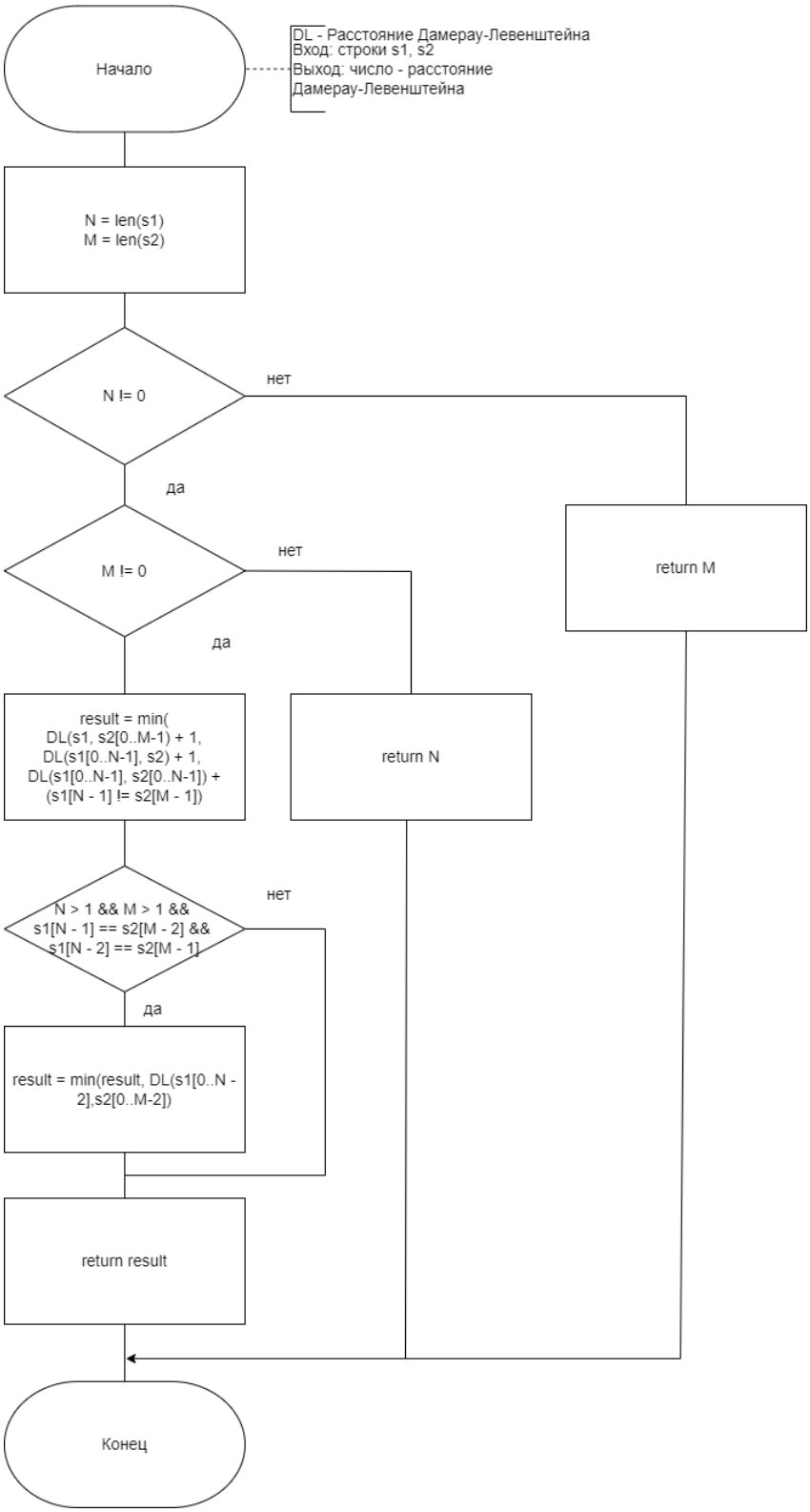

Рисунок 2.3 – Схема рекурсивного алгоритма поиска расстояния Дамерау — Левенштейна

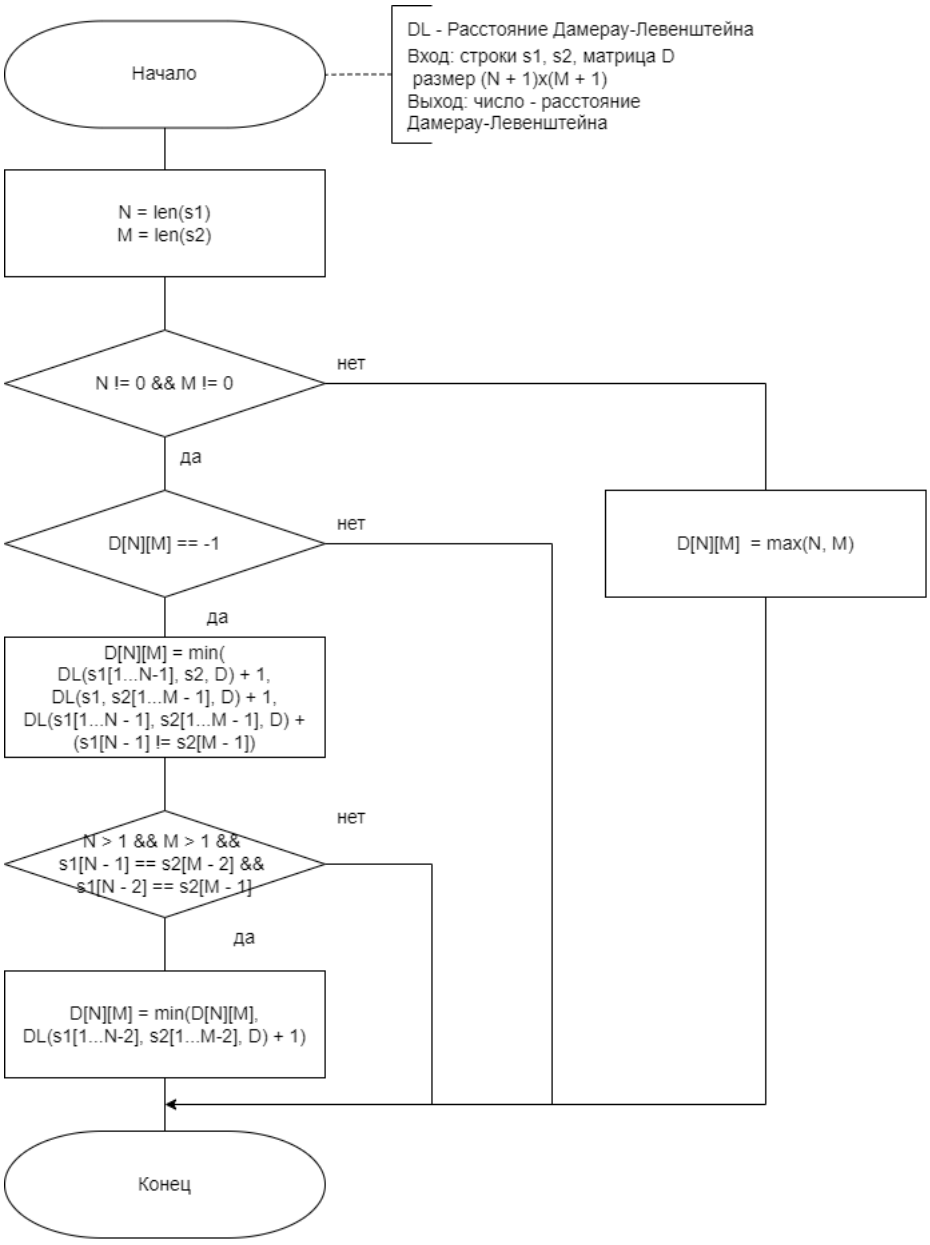

Рисунок 2.4 – Схема рекурсивного алгоритма поиска расстояния Дамерау — Левенштейна с кешем

Вывод

Перечислены требования программе, а также на основе теортических данных, полученных из аналитического раздела, были построены схемы требуемых алгоритмов.

3 Технологическая часть

В данном разделе будут приведены средства реализации, листинги кода и функциональные тесты.

3.1 Средства реализации

Для реализации алгоритмов был выбран язык программирования *Python*, а в качестве среды разработки - *Visual Studio Code*. Для замеров процессорного времени использовалась функция *process_time()* из библиотеки *time*, а для построения графиков - библиотека *matplotlib* [5].

3.2 Сведения о модулях программы

Программа состоит из 4 модулей:

— main.py — точка входа;

— algorythms.py — хранит реализацию алгоритмов;

— compare.py — хранит реализацию системы замера времени;

— constant.py — хранит константы.

3.3 Реализация алгоритмов

Ниже представлены реализации алгоритмов поиска расстояний Левенштейна и Дамерау — Левенштейна (листинги 3.1–3.4).

Листинг 3.1 – Итерационный алгоритм поиска расстояния Левенштейна

```
1 def DLevenshtein(str1, str2):
2     n = len(str1)
3     m = len(str2)
4     matrix = createMatrix(n + 1, m + 1, 0)
5
```

```
    for i in range(0, n + 1):
        for j in range(0, m + 1):
            if (i == 0 or j == 0):
                matrix[i][j] = max(i, j)
            else:
                matrix[i][j] = min(matrix[i - 1][j] + 1,
                                   matrix[i][j - 1] + 1,
                                   matrix[i - 1][j - 1] +
                                    int(str1[i - 1] != str2[j - 1]))

    return matrix
```

Листинг 3.2 – Итерационный алгоритм поиска расстояния Дамерау — Левенштейна

```
def DDamerauLevenshtein(str1, str2):
    n = len(str1)
    m = len(str2)
    matrix = createMatrix(n + 1, m + 1, 0)

    for i in range(0, n + 1):
        for j in range(0, m + 1):
            if (i != 0 and j != 0):
                matrix[i][j] = min(matrix[i - 1][j] + 1,
                        matrix[i][j - 1] + 1,
                        matrix[i - 1][j - 1] +
                            int(str1[i - 1] != str2[j - 1]))
            if (i > 1 and j > 1 and
                str1[i - 1] == str2[j - 2] and
                str1[i - 2] == str2[j - 1]):
                    matrix[i][j] = min(matrix[i][j],
                                       matrix[i - 2][j - 2] + 1)
            else:
                matrix[i][j] = max(i, j)

    return matrix
```

Листинг 3.3 – Рекурсивный алгоритм поиска расстояния Дамерау — Левенштейна

```
def DDamerauLevenshteinRecursive(str1, str2):
```

```
2     n = len(str1)
3     m = len(str2)
4
5     if n != 0:
6         if m != 0:
7             add = DDamerauLevenshteinRecursive(str1,
                  str2[:m-1]) + 1
8             delete = DDamerauLevenshteinRecursive(str1[:n-1],
                  str2) + 1
9             change = DDamerauLevenshteinRecursive(str1[:n-1],
10                        str2[:m-1]) + int(str1[n- 1] != str2[m
                              - 1])
11
12            result = min(add, delete, change)
13
14            if (n >= 2 and m >=2 and
15                str1[n-1] == str2[m-2] and
16                str1[n-2] == str2[m-1]):
17                    result = min(result,
                          DDamerauLevenshteinRecursive(str1[:n-2],
18                    str2[:m-2]) + 1)
19        else:
20            return n
21    else:
22        return m
23    return result
```

Листинг 3.4 – Рекурсивный алгоритм поиска расстояния Дамерау — Левенштейна с кешем

```
1
2 def Recursive(str1, str2, n, m, matrix):
3     if (n != 0 and m != 0):
4         if (matrix[n][m] == -1):
5             delete = Recursive(str1, str2, n - 1, m, matrix) + 1
6             insert = Recursive(str1, str2, n, m - 1, matrix) + 1
7             change = Recursive(str1, str2, n - 1,
8                       m - 1, matrix) +
9                       int(str1[n - 1] != str2[m - 1])
10
11        matrix[n][m] = min(insert, delete, change)
12        if (n > 1 and m > 1 and str1[n - 1] == str2[m - 2] and
```

```
13                 str1[n - 2] == str2[m - 1]):
14                     matrix[n][m] = min(matrix[n][m],
15                         Recursive(str1, str2, n - 2, m - 2, matrix)
                               + 1)
16         else:
17             matrix[n][m] = max(n, m)
18
19         return matrix[n][m]
20
21 def DDamerauLevenshteinRecurciveCache(str1, str2):
22         n = len(str1)
23         m = len(str2)
24
25         matrix = createMatrix(n + 1, m + 1, -1)
26
27         Recursive(str1, str2, n, m, matrix)
28
29         return matrix
```

3.4 Функциональные тесты

В таблице 3.1 приведены функциональные тесты для алгоритмов поиска расстояний Левенштейна и Дамерау — Левенштейна. Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты

	Входные данные		**Выходные данные**			
	str_1	**str_2**	**Левенштейн**	**Дамерау — Левенштейн**		
				Нерекурсивный	**Рекурсивный**	
					Без кэша	**С кэшем**
1			0	0	0	0
2		dang	4	4	4	4
3	apple		5	5	5	5
4	water	water	0	0	0	0
5	apple	apples	1	1	1	1
6	appleses	pallesce	5	4	4	4
7	adaz	diz	2	2	2	2
8	visao	asoiv	5	5	5	5

Вывод

В этом разделе были рассмотрены средства реализации задания, представлены листинги реализации алгоритмов поиска расстояний Левенштейна и Дамерау — Левенштейна, а также тесты.

4 Исследовательская часть

В данном разделе будут представлены примеры работы программы, проведены замеры процессорного времени и предоставлена информация о технических характеристиках устройства.

4.1 Технические характеристики устройства

Ниже представлены характеристики компьютера, на котором проводились замеры времени работы реализации алгоритмов:

— операционная система Windows 10 Домашняя 21H2;

— оперативная память 12 Гб;

— процессор Intel(R) Core(TM) i7-9750H CPU @ 2.6Гц.

Загруженность компонентов:

— процессор - 27%;

— оперативная память - 53%

4.2 Примеры работы программы

На рисунках 4.1-4.2 представлен результат работы программы. В каждом примере пользователем введены две строки и получены результаты вычислений редакционных расстояний.

```
Меню

 1. Расстояние Левенштейна
 2. Расстояние Дамерау-Левенштейна
 3. Росстояние Дамерау-Левенштейна (рекурсивно)
 4. Расстояние Дамерау-Левенштейна (рекурсивно с кешем)
 5. Измерить время
 0. Выход

 Выбор: 2

Введите 1-ую строку:    appleses
Введите 2-ую строку:    pallesce

Матрица, с помощью которой происходило вычисление расстояния Дамерау-Левенштейна:

   Ø  p  a  l  l  e  s  c  e
Ø  0  1  2  3  4  5  6  7  8
a  1  1  1  2  3  4  5  6  7
p  2  1  1  2  3  4  5  6  7
p  3  2  2  2  3  4  5  6  7
l  4  3  3  2  2  3  4  5  6
e  5  4  4  3  3  2  3  4  5
s  6  5  5  4  4  3  2  3  4
e  7  6  6  5  5  4  3  3  3
s  8  7  7  6  6  5  4  4  4

Результат:  4
```

Рисунок 4.1 – Пример работы программы (1)

```
Меню

 1. Расстояние Левенштейна
 2. Расстояние Дамерау-Левенштейна
 3. Росстояние Дамерау-Левенштейна (рекурсивно)
 4. Расстояние Дамерау-Левенштейна (рекурсивно с кешем)
 5. Измерить время
 0. Выход

 Выбор: 3

Введите 1-ую строку:    flksdfj
Введите 2-ую строку:    ewrlwer

Результат:  7
```

Рисунок 4.2 – Пример работы программы (2)

4.3 Время выполнения реализации алгоритмов

Для замера процессорного времени выполнения реализации алгоритмов использовалась функция *process_ time()* библиотеки *time*. Возвращаемый результат - время в миллисекундах, число типа *float*. Чтобы получить достаточно точное значение, производилось усреднение времени. Количество запусков замера процессорного времени 10000 раз.

В замерах использовались строки длиной от 0 до 7 символов, длины строк совпадают, строки заполняются случайными символами. Количество повторных измерений времени варьировалось от 50 до 1000. в зависимости от длин входных строк.

Таблица 4.1 – Результаты замеров процессорного времени

Длина	**Выходные данные**			
	Левенштейн	**Дамерау — Левенштейн**		
		Нерекурсивный	**Рекурсивный**	
			Без кэша	**С кэшем**
0	0	0.01562	0	0.01562
1	0.03125	0.03125	0.03125	0
2	0.0625	0.04688	0.0625	0.0625
3	0.0625	0.09375	0.28125	0.10938
4	0.10938	0.10938	1.45312	0.17188
5	0.14062	0.15625	7.78125	0.26562
6	0.21875	0.23438	42.35938	0.35938
7	0.28125	0.29688	216.6875	0.46875

На рисунках 4.3–4.5 также приведены результаты замеров процессорного времени.

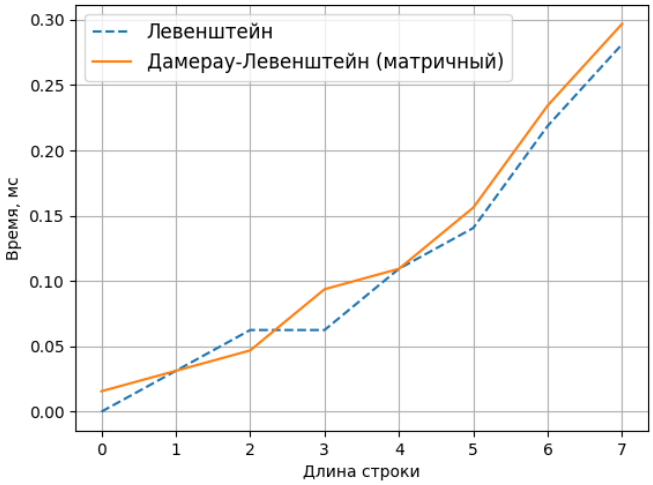

Рисунок 4.3 – Зависимость времени выполнения реализации итерационных алгоритмов поиска расстояний Левенштейна от длины строки

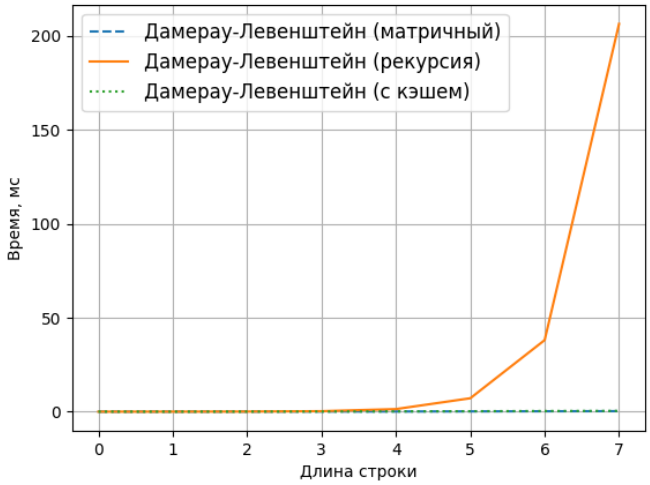

Рисунок 4.4 – Зависимость времени выполнения реализации алгоритмов поиска расстояний Дамерау — Левенштейна от длины строки

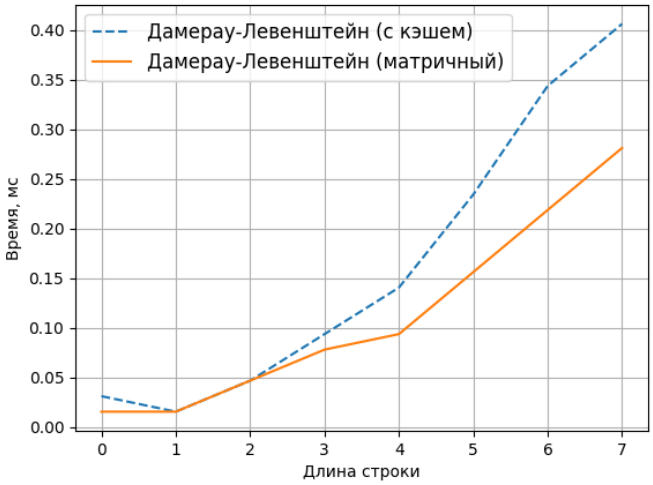

Рисунок 4.5 – Зависимость времени выполнения реализации итерационного алгоритма поиска расстояний Дамерау — Левенштейна и рекурсивного с кешем от длины строки

Вывод: Рекурсивный алгоритм поиска расстояния Дамерау — Левенштейна работает дольше всех; итерационный алгоритмов поиска расстояний Левенштейна работает бытсрее всехю

4.4 Использование памяти

Затраты по памяти для реализвции алгоритмов поиска расстояния Левенштейна (итерационного) и Дамерау — Левенштейна (итерационного) не отличаются друг от друга:

— матрица: $(n + 1) \cdot (m + 1) \cdot \text{sizeof(int)}$;

— строки str_1, str_2: $(n + m + 2) \cdot \text{sizeof(char)}$;

— длины строк n, m: $2 \cdot \text{sizeof(int)}$;

— дополнительные переменные (i, j): $2 \cdot \text{sizeof(int)}$;

— адрес возврата.

Итого:

$$(n + 1)(m + 1) \cdot \text{sizeof(int)} + (n + m + 2) \cdot \text{sizeof(char)} + 4 \cdot \text{sizeof(int)} \quad (4.1)$$

Затраты по памяти для алгоритма поиска расстояния Дамерау — Левенштейна (рекурсивный), для одного вызова:

— строки str_1, str_2: $(n + m + 2) \cdot \text{sizeof(char)}$;

— длины строк n, m: $2 \cdot \text{sizeof(int)}$;

— дополнительные переменные (add, delete, change, result): $4 \cdot \text{sizeof(int)}$;

— адрес возврата.

K - максимальное количество вызовов рекурсии:

$$K = M + N \tag{4.2}$$

Итого:

$$((n + m + 2) \cdot \text{sizeof(char)} + 6 \cdot \text{sizeof(int)}) \cdot (M + N) \tag{4.3}$$

Затраты по памяти для алгоритма поиска расстояния Дамерау — Левенштейна (рекурсивный с кешем), для одного вызова:

— строки str_1, str_2: $(n + m + 2) \cdot \text{sizeof(char)}$;

— длины строк n, m: $2 \cdot \text{sizeof(int)}$;

— дополнительные переменные (add, delete, change, result): $4 \cdot \text{sizeof(int)}t$;

— адрес возврата;

и матрица - $(n + 1) \cdot (m + 1) \cdot \text{sizeof(int)}$.

K - максимальное количество вызовов рекурсии:

$$K = M + N \tag{4.4}$$

Итого:

$$((n+m+2) \cdot \text{sizeof(char)} + 6 \cdot \text{sizeof(int)}) \cdot (M+N) + (n+1) \cdot (m+1) \cdot \text{sizeof(int)} \tag{4.5}$$

На рисунках 4.6 также приведены результаты замеров памяти.

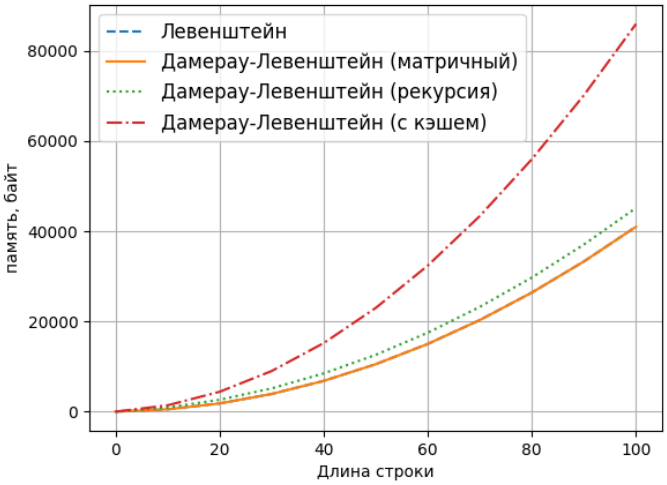

Рисунок 4.6 – Замеры памяти, которую затрачивают реализации алогритмов

4.5 Вывод

В результате замеров процессорного времени выделены следущие аспекты:

— в результате эксперимента было получено, что при длине строк в более 4 символов, алгоритм Левенштейна быстрее Дамерау — Левенштейна в 16%;

— рекурсивный алгоритм поиска расстояния Дамерау — Левенштейна уже при длине строк, равной 4, символа проигрывает в 15 раз по времени итерационной и в 10 раз рекурсивной с кешем реализациям;

— итерационный алгоритм поиска расстояний Дамерау — Левенштейна в среднем на 49 % – 54 % быстрее рекурсивного с кешем для длин строк от 0 до 7 символов;

Проведя анализ оценки затрат реализаций алгоритмов по памяти, можно сказать, что рекурсивные алгоритмы менее затратны, так как для них максимальный размер памяти растет прямо пропорционально сумме длин строк.

Заключение

Был изучен метод динамического программирования на материале алгоритмов Левенштейна и Дамерау — Левенштейна. Также описаны алгоритмы Левенштейна и Дамерау — Левенштейна нахождения расстояния между строками, получены практические навыки реализации указанных алгоритмов в матричной и рекурсивных версиях.

Решены все поставленные задачи:

— изучены, разработаны и реализованы алгоритмы поиска расстояний Левенштейна и Дамерау — Левенштейна;

— выполнены замеры алгоритмов процессорного времени работы реализаций алгоритмов;

— проведен сравнительный анализ нерекурсивных алгоритмов поиска расстояний Левенштейна и Дамерау — Левенштейна;

— проведен сравнительный анализ трех алгоритмов поиска расстояний Дамерау — Левенштейна.

— подготовлен отчет о лабораторной работе.

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 пришёл к выводу, что алгоритм Левенштейна заметно выигрывает Дамерау — Левенштейна в 16% по времени ри длине строк в более 4, рекурсивный алгоритм Дамерау — Левенштейна уже при длине строк равной 4 символа проигрывает в 15 раз по времени итерационной и в 10 раз рекурсивной с кешем реализациям, итерационный алгоритм поиска расстояний Дамерау — Левенштейна в среднем на 49 % – 54 % быстрее рекурсивного с кешем для длин строк от 0 до 7 символов. Несмотря на то, что итерационные алгоритмы обладают высоким быстродействием, при больших длинах строк они занимают довольно много памяти под матрицу.

Список использованной литературы

1 Левенштейн В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов //. Т. 163. — М.: Издательство «Наука», Доклады АН СССР, 1965. — Гл. 4

2 Расстояние Левенштейна. - URL: https://www.baeldung.com/cs/levenshtein-distance-computation (дата обращения: 22.09.2023)

3 Алгоритм Левенштейна. - URL: https://www.cuelogic.com/blog/the-levenshtein-algorithm (дата обращения: 22.09.2023)

4 Расстояние Дамерау-Левенштейна - URL: https://bmcbioinformatics.biomedcentral.com/articles/10.1186/s12859-019-2819-0/ (дата обращения: 21.09.2023)

5 Matplotlib documentation [Электронный ресурс]. - URL: https://matplotlib.org/stable/index.html (дата обращения: 22.09.2023)