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 Расстояние Левенштейна и Дамерау-Левенштейна

Студент Пронина Л.Ю.

Группа ИУ7-54Б

Оценка (баллы) ____________

Преподаватель Волкова Л. Л.

Москва — 2023 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Редакционное расстояние Левенштейна и Дамерау—Левентшейна представляет собой минимальное количество операций (вставка, удаление и замена символов), требуемое для преобразования одной строки в другую. Эти метрики широко используются в области редактирования текста, биоинформатики, автоматического исправления ошибок и других приложений, где важно оценить разницу между двумя строками.

Целью данной работы является изучение, реализация и исследование алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна. Для достижения поставленной цели необходимо выполнить следующие задачи:

- изучить и реализовать алгоритмы нахождения расстояний Левенштейна и Дамерау-Левенштейна;
- провести тестирование, чтобы измерить время выполнения и использование памяти для каждого алгоритма;
- провести сравнение процессорного времени выполнения алгоритмов и памяти;
- описать и обосновать полученные результаты в отчете о выполненной лабораторной работе.

1 Аналитическая часть

В данном разделе будут разобраны алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна.

1.1 Расстояние Левенштейна

Расстояние Левенштейна — метрика, измеряющая разность между двумя последовательностями символов. Расстояние Левенштейна — это минимальное количество редакторских операций вставки (I, от англ. insert), замены (R, от англ. replace) и удаления (D, от англ. delete), необходимых для преобразования одной строки в другую[1].

Стоимости операций могут зависеть от вида операций:

1) $w(a, b)$ — цена замены символа a на b;

2) $w(\lambda, b)$ — цена вставки символа b;

3) $w(a, \lambda)$ — цена удаления символа a.

Будем считать стоимость каждой вышеизложенной операции равной 1:

— $w(a, b) = 1$, $a \neq b$, в противном случае замена не происходит;

— $w(\lambda, b) = 1$;

— $w(a, \lambda) = 1$.

Введем понятие совпадения символов — M (от англ. match). Его стоимость будет равна 0, то есть $w(a, a) = 0$.

Введем в рассмотрение функцию $D(i, j)$, значением которой является редакционное расстояние между подстроками $S_1[1...i]$ и $S_2[1...j]$.

Расстояние Левенштейна между двумя строками S_1 и S_2 длиной M и

N соответственно рассчитывается по рекуррентной формуле (1.1).

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

где сравнение символов строк S_1 и S_2 рассчитывается как:

$$m(a,b) = \begin{cases} 0 & \text{если a = b}, \\ 1 & \text{иначе}. \end{cases} \tag{1.2}$$

1.1.1 Нерекурсивный алгоритм нахождения расстояния Левенштейна

Рекурсивная реализация алгоритма Левенштейна малоэффективна по времени при больших M и N из-за повторного вычисления одних и тех же промежуточных результатов. Для оптимизации можно использовать итерационную реализацию заполнения матрицы промежуточными значениями $D(i,j)$.

В качестве структуры данных для хранения промежуточных значений можно использовать матрицу, имеющую размеры:

$$(N+1) \times (M+1) \tag{1.3}$$

Значения в ячейке $[i,j]$ равно значению $D(S1[1...i], S2[1...j])$. Первый элемент матрицы заполнен нулем. Всю таблицу заполнять в соответствии с формулой (1.1).

Однако матричный алгоритм является малоэффективным по памяти по сравнению с рекурсивным при больших M и N, т.к. множество промежуточных значений $D(i, j)$ хранится в памяти после их использования. Для оптимизации по памяти рекурсивного алгоритма нахождения расстояния Левенштейна можно использовать кеш, т.е. пару строк, содержащую значения $D(i, j)$, вычисленные в предыдущей итерации, алгоритма и значения $D(i, j)$, вычисляемые в текущей итерации.

1.2 Расстояние Дамерау-Левенштейна

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 трем базовым операциям добавляется операция транспозиции T (от англ. transposition).

Расстояние Дамерау-Левенштейна может быть вычислено по рекуррентной формуле:

$$D(i,j) = \begin{cases} 0, & \text{если i = 0 и j = 0,} \\ i, & \text{если j = 0 и i > 0,} \\ j, & \text{если i = 0 и j > 0,} \\ min\{D(i,j-1)+1, & \\ D(i-1,j)+1, & \text{если i > 0 и j > 0,} \\ D(i-1,j-1)+m(S_1[i],S_2[j]), & \text{если } S_1[i] = S_2[j-1], \\ D(i-2,j-2)+1, & \text{если } S_1[i-1] = S_2[j], \\ \}, & \\ min\{D(i,j-1)+1, & \\ D(i-1,j)+1, & \text{иначе,} \\ D(i-1,j-1)+m(S_1[i],S_2[j]), & \\ \}. & \end{cases} \tag{1.4}$$

1.2.1 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна

Рекурсивный алгоритм реализует формулу (1.4), функция D составлена таким образом, что верно следующее.

1) Для передачи из пустой строки в пустую требуется ноль операций.

2) Для перевода из пустой строки в строку a требуется $|a|$ операций.

3) Для перевода из строки a в пустую строку требуется $|a|$ операций.

4) Для перевода из строки a в строку b требуется выполнить последовательно некоторое количество операций удаления, вставки, замены, транспозиции в некоторой последовательности. Последовательность поведения любых двух операций можно поменять, порядок поведения операций не имеет никакого значения. Если полагать, что *a', b'* — строки a и b без последнего символа соответственно, а *a", b"* — строки a и b без двух последних символов, то цена преобразования из строки a в b выражается из элементов, представленных ниже:

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

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

   — сумма цены преобразования из a' в b' и операции замены, предполагая, что a и b оканчиваются на разные символы;

   — сумма цены преобразования из a'' в b'' и операции перестановки, предполагая, что длины a'' и b'' больше 1 и последние два символа a'', поменянные местами, совпадут с двумя последними символами b'';

   — цена преобразования из a' в b', предполагая, что a и b оканчиваются на один и тот же символ.

Минимальной стоимостью преобразования будет минимальное значение приведенных вариантов.

1.2.2 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна с кешированием

Рекурсивная реализация алгоритма Дамерау-Левенштейна малоэффективна по времени при больших M и N по причине проблемы повторных вычислений значений расстояний между подстроками. Для оптимизации алгоритма нахождения расстояния Левенштейна можно использовать матрицу в целях хранения соответствующих промежуточных значений. В таком случае алгоритм представляет собой рекурсивное заполнение матрицы $A_{|a|,|b|}$ промежуточными значениями $D(i, j)$, такое хранение промежуточных данных можно назвать кешем для рекурсивного алгоритма.

1.2.3 Нерекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна

Рекурсивная реализация алгоритма Левенштейна с кешированием малоэффективна по времени при больших M и N. Для оптимизации можно использовать итерационную реализацию заполнения матрицы промежуточными значениями $D(i, j)$.

В качестве структуры данных для хранения промежуточных значений можно использовать *матрицу*, имеющую размеры:

$$(N + 1) \times (M + 1), \tag{1.5}$$

Значение в ячейке $[i, j]$ равно значению $D(S1[1...i], S2[1...j])$. Первый элемент заполнен нулем. Всю таблицу заполняем в соответствии с формулой (1.4).

1.3 Вывод

В данном разделе были даны определения расстояний Левенштейна и Дамерау-Левенштейна, а также рассмотрены 4 алгоритма вычисления указанных расстояний.

2 Конструкторская часть

В этом разделе будет представлены схемы алгоритмов вычисления расстояния Левенштейна и Дамерау-Левенштейна.

2.1 Сведения о модулях программы

Программа состоит из двух модулей:

- *main.py* — основной файл программы, в котором находится главный код, выполняющийся при запуске программы;
- *algorythms.py* — файл, содержащий код всех алгоритмов.

2.2 Разработка алгоритмов

На рисунках 2.1-2.4 представлены схемы алгоритмов вычисления расстояния Левенштейна и Дамерау-Левенштейна.

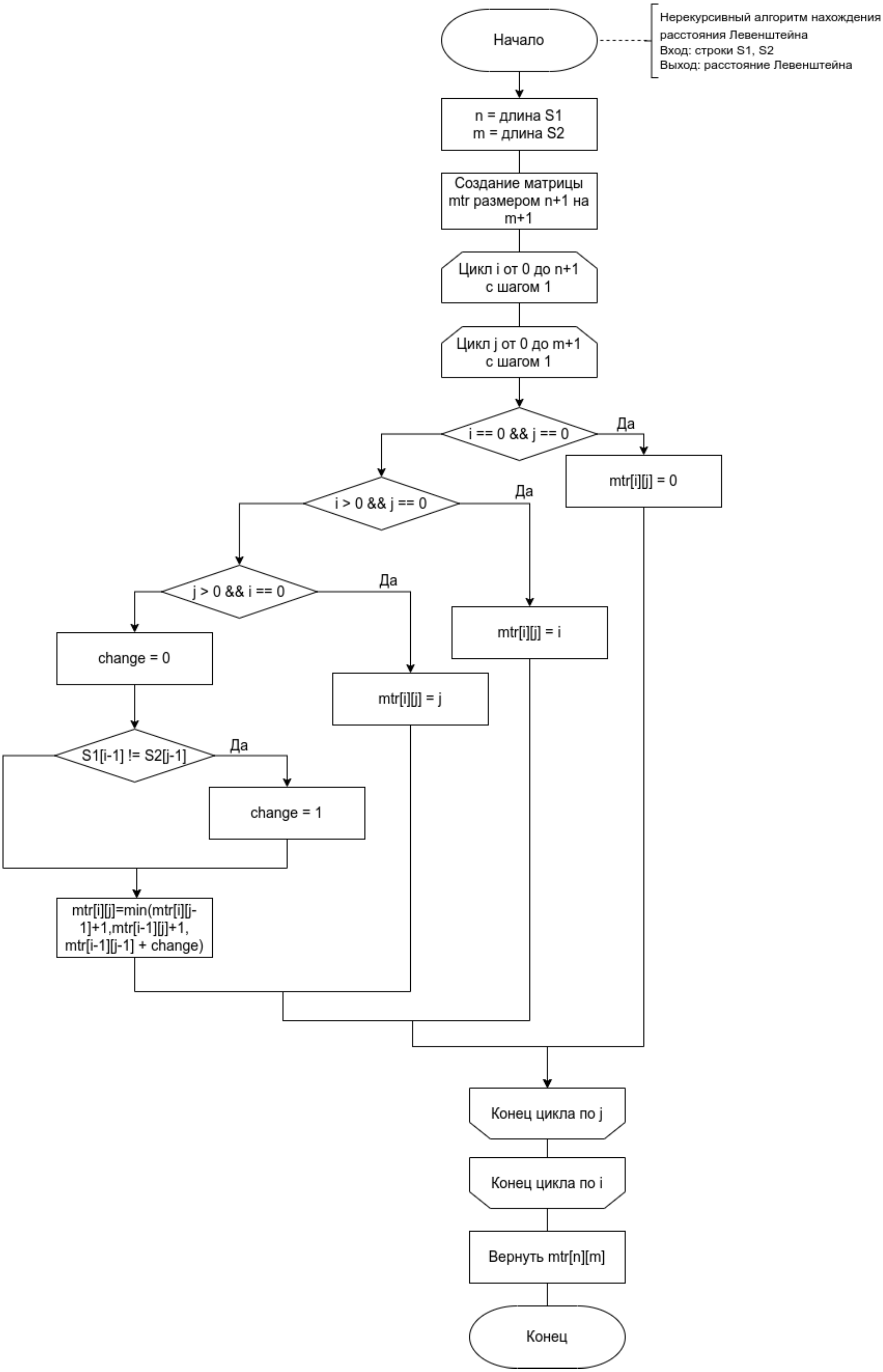


Рисунок 2.1 – Схема матричного алгоритма нахождения расстояния Левенштейна

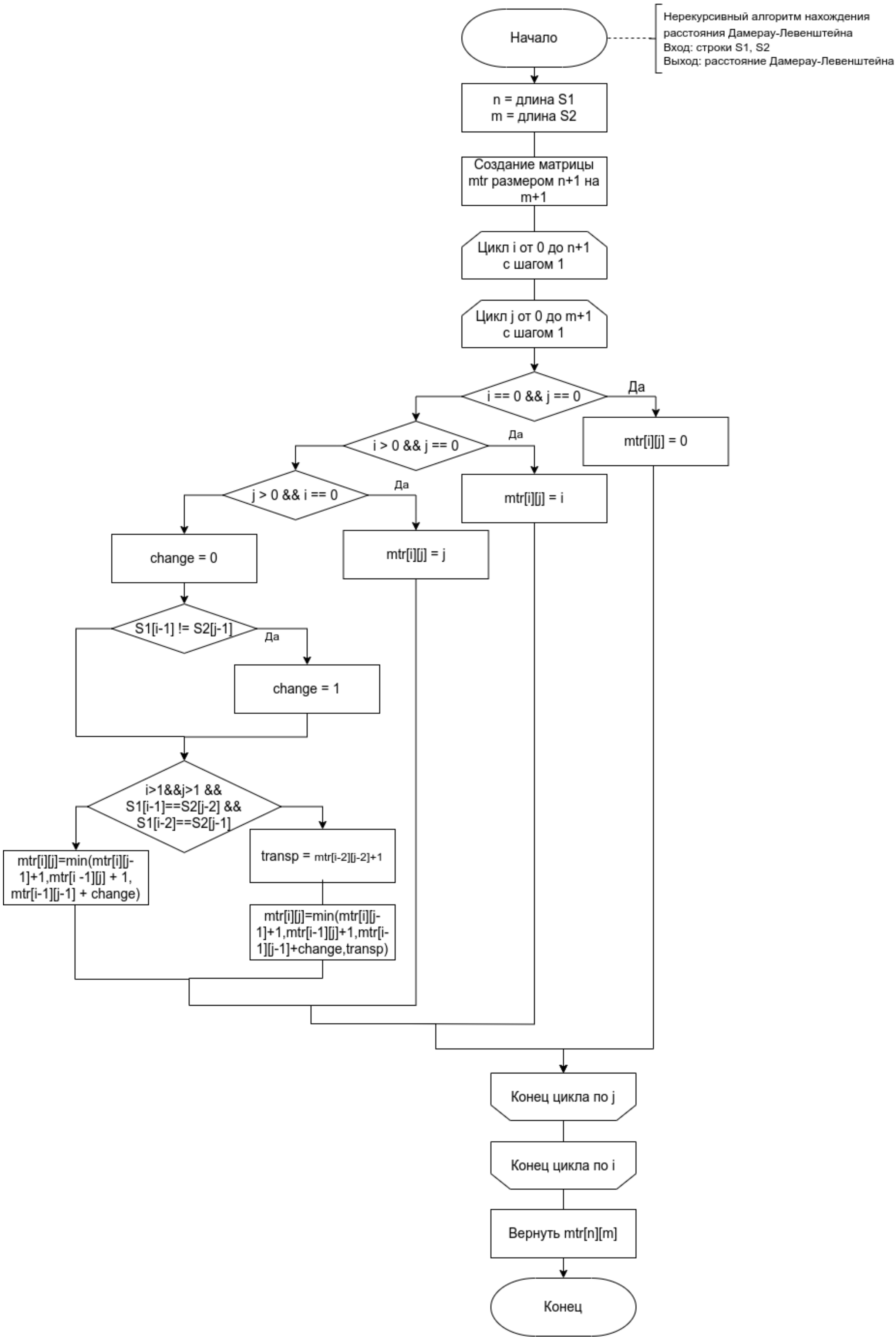


Рисунок 2.2 – Схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна

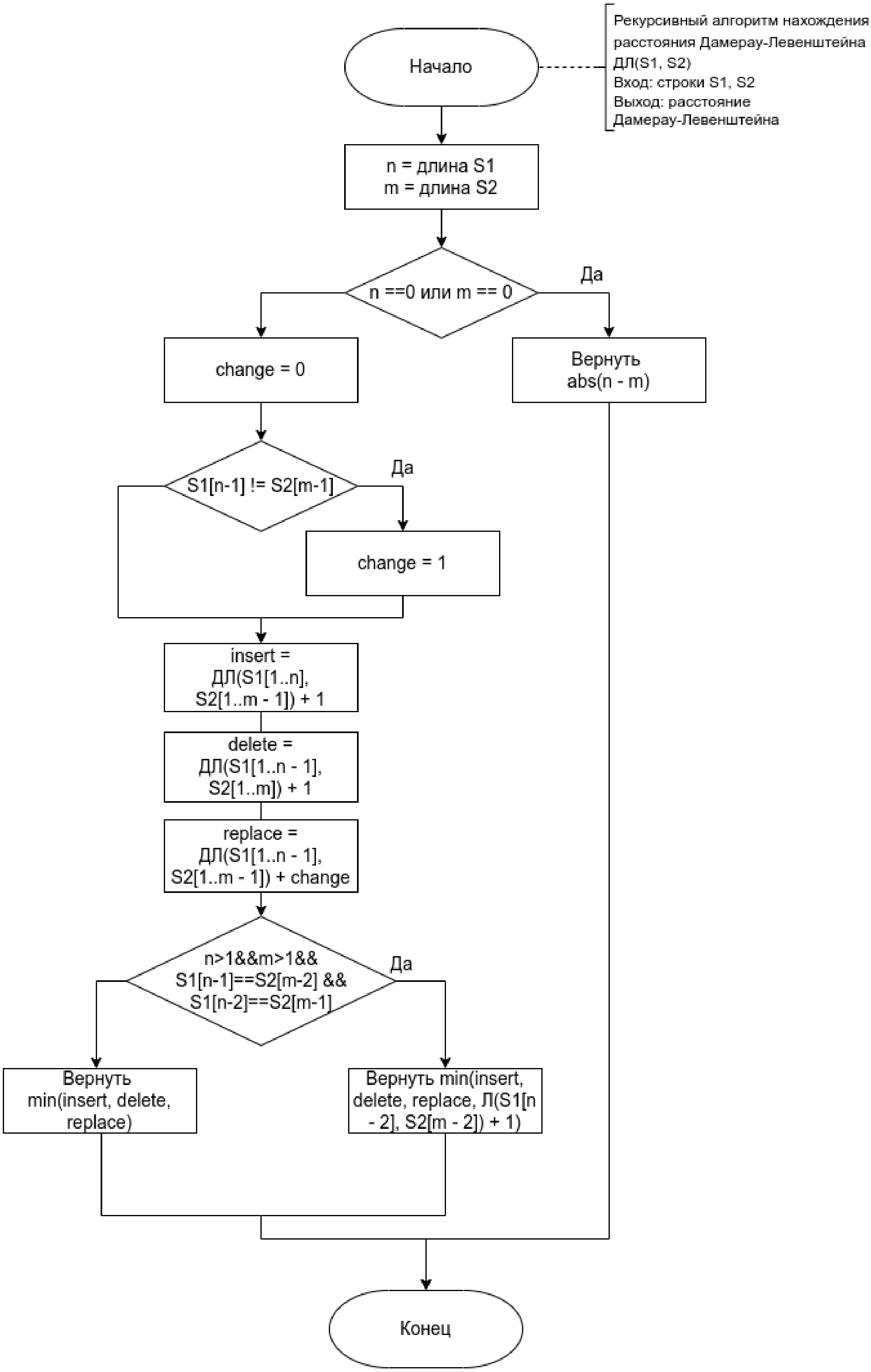


Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

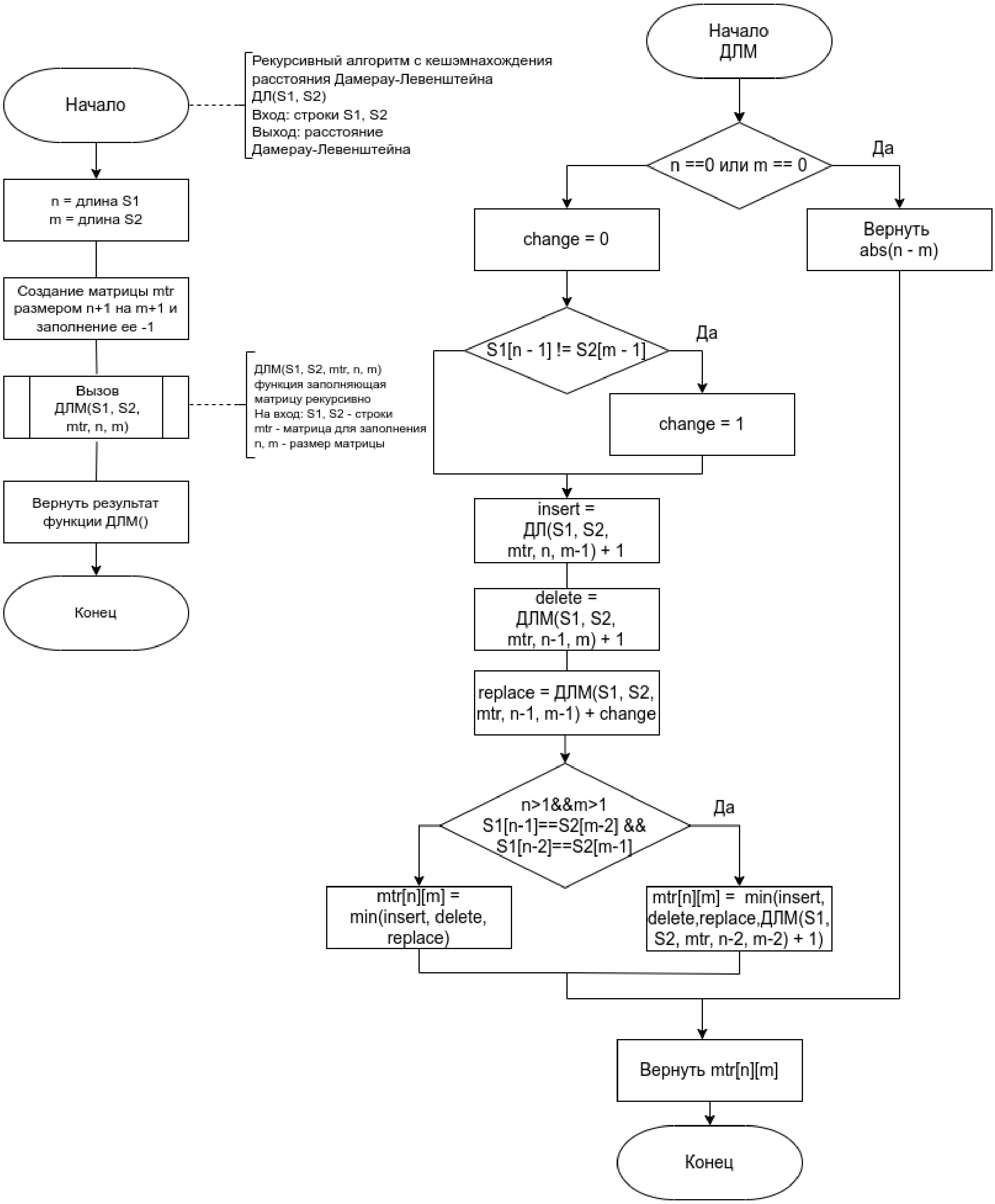


Рисунок 2.4 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с использованием кеша

2.3 Вывод

В данном разделе были представлены схемы алгоритмов, рассматриваемых в лабораторной работе.

3 Технологическая часть

В данном разделе будут рассмотрены средства реализации, а также представлены листинги алгоритмов определения расстояния Левенштейна и Дамерау-Левенштейна.

3.1 Средства реализации

В данной работе для реализации был выбран язык программирования *Python*[2]. В текущей лабораторной работе требуется замерить процессорное время для выполняемой программы, а также построить графики. Все эти инструменты присутствуют в выбранном языке программирования.

Время работы было замерено с помощью функции *process_time()* из библиотеки *time*[3].

3.2 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие структуры данных:

- две строки типа *str*;
- длина строки — целое число типа *int*;
- в матричных реализациях алгоритмов Левенштейна и Дамерау-Левенштейна дополнительно используется матрица, которая является двумерным списком типа *int*.

3.3 Реализация алгоритмов

В листингах 3.1-3.4 представлены реализации алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

Листинг 3.1 – Алгоритм нахождения расстояния Левенштейна (матричный)

```
def levenshtein_matrix(str1, str2):
    m = len(str1)
    n = len(str2)
    matrix = [[0] * (m + 1) for _ in range(n + 1)]
    for i in range(n + 1):
        matrix[i][0] = i
    for j in range(m + 1):
        matrix[0][j] = j
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            cost = 0 if str1[i - 1] == str2[j - 1] else 1
            matrix[i][j] = min(matrix[i - 1][j] + 1,
            matrix[i][j - 1] + 1,
            matrix[i - 1][j - 1] + cost)
    return matrix[m][n]
```

Листинг 3.2 – Алгоритм нахождения расстояния Дамерау-Левенштейна

```
def damerau_levenshtein_matrix(str1, str2):
    m = len(str1)
    n = len(str2)
    matrix = [[0] * (m + 1) for _ in range(n + 1)]
    for i in range(n + 1):
        matrix[i][0] = i
    for j in range(m + 1):
        matrix[0][j] = j
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            cost = 0 if str1[i - 1] == str2[j - 1] else 1
            matrix[i][j] = min(matrix[i - 1][j] + 1,
            matrix[i][j - 1] + 1,
            matrix[i - 1][j - 1] + cost)
    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)
    return matrix[m][n]
```

Листинг 3.3 – Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна

```
def damerau_levenshtein_recursive(str1, str2):
    if len(str1) == 0:
        return len(str2)
    if len(str2) == 0:
        return len(str1)
    flag = 0 if (str1[-1] == str2[-1]) else 1
    add = damerau_levenshtein_recursive(str1[:-1], str2) + 1
    delete = damerau_levenshtein_recursive(str1, str2[:-1]) + 1
    change = damerau_levenshtein_recursive(str1[:-1], str2[:-1]) +
        flag
    transposition = damerau_levenshtein_recursive(str1[:-2],
        str2[:-2]) + 1
    if (len(str1) > 1 and len(str2) > 1 and str1[-1] == str2[-2]
        and str1[-2] == str2[-1]):
        minimum  = min(add, delete, change, transposition)
    else:
        minimum = min(add, delete, change)
    return minimum
```

Листинг 3.4 – Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна с использованием кеша

```
def damerau_levenshtein_cache_recursive(str1, str2):
    cache = {}
    def damerau_levenshtein_recursive(str1, str2):
        if (str1, str2) in cache:
            return cache[(str1, str2)]
        if len(str1) == 0:
            return len(str2)
        if len(str2) == 0:
            return len(str1)
        deletion = damerau_levenshtein_recursive(str1[:-1], str2) + 1
        insertion = damerau_levenshtein_recursive(str1, str2[:-1]) + 1
        substitution = damerau_levenshtein_recursive(str1[:-1], str2[:-1]) + (str1[-1] != str2[-1])
        transposition = float('inf')
        if len(str1) > 1 and len(str2) > 1 and str1[-1] == str2[-2] and str1[-2] == str2[-1]:
            transposition = damerau_levenshtein_recursive(str1[:-2], str2[:-2]) + 1
        result = min(deletion, insertion, substitution, transposition)
        cache[(str1, str2)] = result
        return result
    distance = damerau_levenshtein_recursive(str1, str2)
    return distance
```

3.4 Классы эквивалентности тестирования

Для тестирования выделены классы эквивалентности, представленные ниже.

1) Две пустые строки.

2) Одна из строк пустая.

3) Расстояния, которые вычислены алгоритмами Левенштейна и Дамерау-Левенштейна, равны.

4) Расстояния, которые вычислены алгоритмами Левенштейна и Дамерау-Левенштейна, не равны.

3.5 Функциональные тесты

В таблице 3.1 приведены тесты для функций, реализующих алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна. Тесты *для всех алгоритмов* пройдены успешно.

Таблица 3.1 – Функциональные тесты

			Ожидаемый результат	
№	Строка 1	Строка 2	Левенштейн	Дамерау-Л.
1			0	0
2		слово	5	5
3	слова		5	5
4	кот	ком	1	1
5	парк	прак	2	1
6	машина	сирена	4	4
7	здравствуйте	до свидания	10	10
8	кошка	собака	3	3
9	карта	карат	2	1
10	спать	встать	2	2
11	пока	привет	5	5
12	abcdef	bacfe	4	3

3.6 Вывод

Были представлены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна, которые были описаны в предыдущем разделе. Также была приведена информация о выбранных средствах для разработки алгоритмов.

4 Исследовательская часть

В данном разделе будут приведены примеры работы программы, а также проведен сравнительный анализ алгоритмов при различных ситуациях на основе полученных данных.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование представлены далее:

- операционная система: Ubuntu 22.04.3 [4] Linux [5] x86_64;
- память: 16 Гб;
- процессор: Intel® Core™ i5-1135G7 @ 2.40Гц.

При тестировании ноутбук не был включен в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, а также системой тестирования.

4.2 Демонстрация работы программы

На рисунках 4.1, 4.2 представлен результат работы программы.

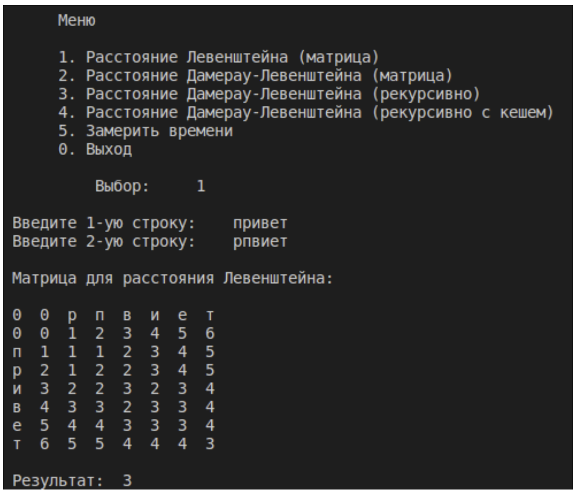
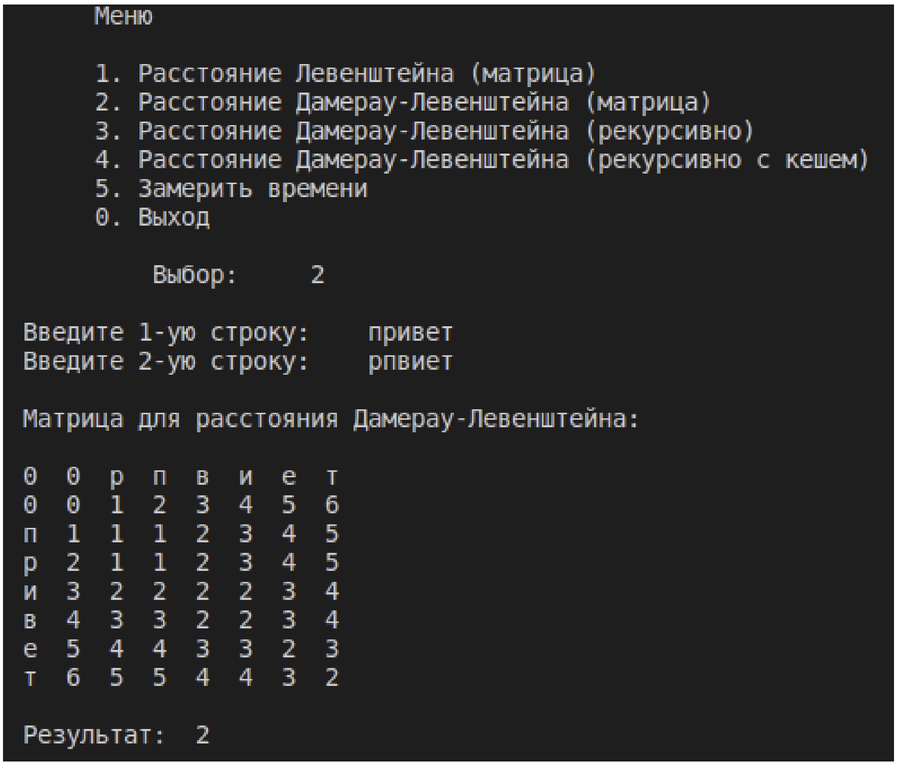


Рисунок 4.1 – Пример работы программы


```
    Меню

    1. Расстояние Левенштейна (матрица)
    2. Расстояние Дамерау-Левенштейна (матрица)
    3. Расстояние Дамерау-Левенштейна (рекурсивно)
    4. Расстояние Дамерау-Левенштейна (рекурсивно с кешем)
    5. Замерить времени
    0. Выход

        Выбор:     2

Введите 1-ую строку:    привет
Введите 2-ую строку:    рпвиет

Матрица для расстояния Дамерау-Левенштейна:

0  0  р  п  в  и  е  т
0  0  1  2  3  4  5  6
п  1  1  1  2  3  4  5
р  2  1  1  2  3  4  5
и  3  2  2  2  2  3  4
в  4  3  3  2  2  3  4
е  5  4  4  3  3  2  3
т  6  5  5  4  4  3  2

Результат:  2
```

Рисунок 4.2 – Пример работы программы

4.3 Время выполнения алгоритмов

Как было сказано выше, используется функция замера процессорного времени process_time() из библиотеки time на Python. Функция возвращает пользовательское процессорное время типа float.

Функция используется дважды: перед началом выполнения алгоритма и после завершения, затем из конечного времени вычитается начальное, чтобы получить результат.

Замеры проводились по 100 раз на различных входных данных для длины слова от 1 до 10 в случае рекурсивного алгоритма Дамерау-Левенштейна и от 1 до 100 для остальных алгоритмов.

Результаты замеров приведены в таблице 4.1 (время в мс). Время для рекурсивной реализации алгоритма Дамерау-Левентшейна измерялось только для длины строки не более 10, так как он работает относительно сильно дольше остальных.

Также на рисунках 4.3, 4.4 приведены графические результаты замеров.

Таблица 4.1 – Результаты замеров времени

Длина	Л.(матр.)	Д.-Л.(матр.)	Д.-Л.(рек.)	Д.-Л.(рек. с кешэм)
1	0.0068	0.0057	0.0012	0.0064
2	0.0092	0.0090	0.0048	0.0148
3	0.0099	0.0120	0.0237	0.0227
4	0.0127	0.0126	0.1229	0.0270
5	0.0133	0.0147	0.6894	0.0321
6	0.0152	0.0182	3.9311	0.0439
7	0.0181	0.0201	22.0361	0.0579
8	0.0204	0.0257	126.6794	0.0754
9	0.0256	0.0317	793.4876	0.0957
10	0.0308	0.0393	4651.9602	0.1208
20	0.1632	0.1921	-	0.7842
30	0.4343	0.4678	-	1.6324
40	0.7198	0.7923	-	2.8983
50	0.9457	1.1793	-	4.5392
60	1.3146	1.7213	-	6.7496
70	1.8647	2.3428	-	9.0544
80	2.3966	3.0905	-	12.2664
90	3.0661	3.8628	-	15.6859
100	3.8084	4.7583	-	19.6995

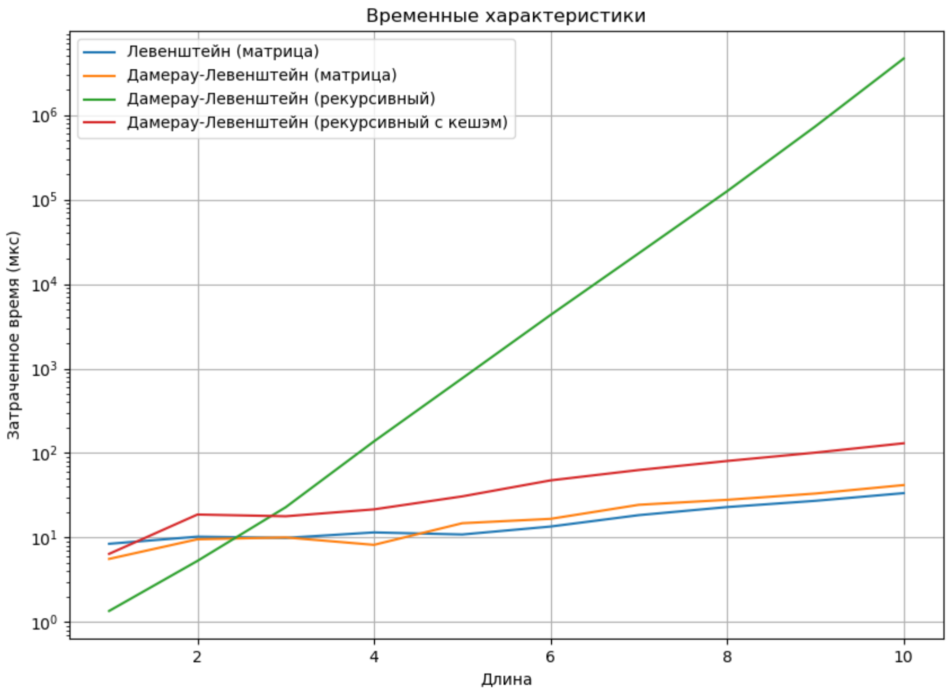


Рисунок 4.3 – Сравнение времени выполнения алгоритмов Левенштейна с использованием матрицы и Дамерау-Левенштейна с использованием матрицы, рекурсивный без кеша и с ним

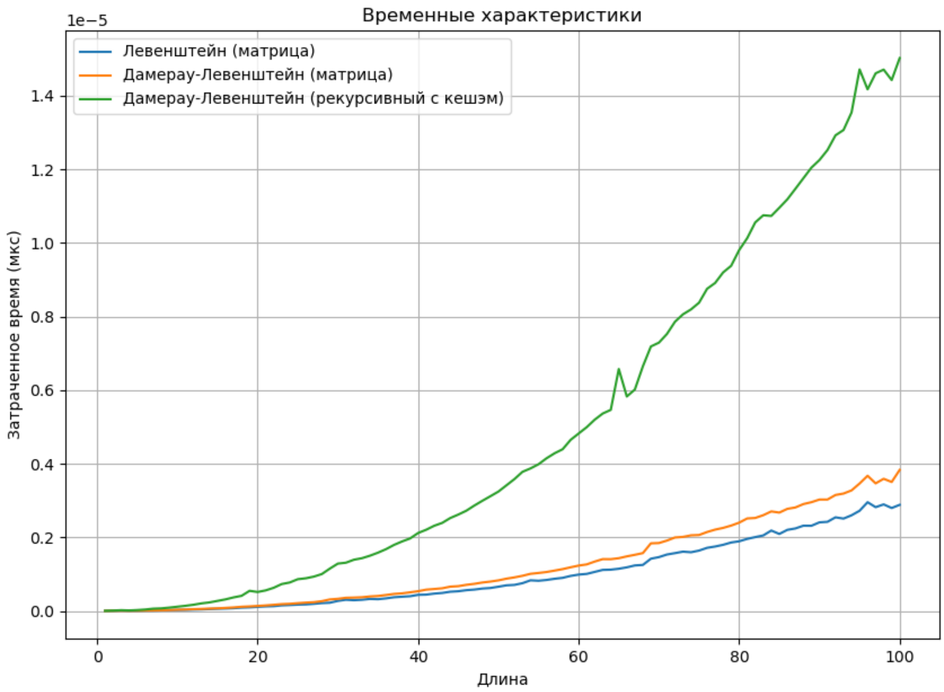


Рисунок 4.4 – Сравнение времени выполнения алгоритмов Левенштейна с использованием матрицы и Дамерау-Левенштейна с использованием матрицы и рекурсивный с использованием кеша

4.4 Использование памяти

Алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна, с точки зрения использования памяти, не отличаются друг от друга. Поэтому достаточно рассмотреть различия между рекурсивной и матричной реализациями этих алгоритмов.

При рекурсивной реализации алгоритма максимальная глубина стека вызовов равна сумме длин входящих строк. Таким образом, максимальный расход памяти можно вычислить с помощью формулы (4.1).

$$(sizeof(S_1) + sizeof(S_2)) \cdot (2 \cdot sizeof(\text{string}) + 2 \cdot sizeof(\text{int}) + sizeof(\text{bool})), \tag{4.1}$$

где $sizeof$ — оператор вычисления размера, S_1, S_2 — строки, int — целочисленный тип, string — строковый тип, bool - логический тип.

Использование памяти при итеративной реализации теоретически вычисляется по формуле (4.2).

$$(sizeof(S_1)+1) \cdot (sizeof(S_2)+1) \cdot sizeof(\text{int}) + 5 \cdot sizeof(\text{int}) + 2 \cdot sizeof(\text{string})) \tag{4.2}$$

4.5 Вывод

В данном разделе было представлено сравнение количества затраченного времени и памяти алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна. Наименее затратным по времени оказался итеративный алгоритм нахождения расстояния Левенштейна, а наиболее затратным — рекурсивный алгоритм Дамерау-Левенштейна.

Исходя из замеров по памяти, итеративные алгоритмы проигрывают рекурсивным, потому что максимальный размер памяти в них растет, как произведение длин строк, а в рекурсивных — как сумма длин строк.

Так как во время печати очень часто возникают ошибки связанные с транспозицией букв, алгоритм поиска расстояния Дамерау-Левенштейна является наиболее предпочтительным, не смотря на то, что он проигрывает по времени и памяти алгоритму Левенштейна.

Также при проведении эксперимента было выявлено, что на длине строк в 4 символа рекурсивная реализация алгоритма Дамерау-Левенштейна уже в 10 раз медленнее матричной реализации. При увеличении длины строк в геометрической прогрессии растет и время работы рекурсивной реализации. Следовательно, стоит использовать матричную реализацию для строк длиной более 4 символов.

ЗАКЛЮЧЕНИЕ

В рамках данной лабораторной работы были изучены, реализованы и протестированы алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна. Также был проведён сравнительный анализ алгоритмов по затраченному процессорному времени и памяти и подготовлен отчет о лабораторной работе.

Экспериментально было определено, что наименее затратным по времени оказался итеративный алгоритм нахождения расстояния Левенштейна, а наиболее затратным — рекурсивный алгоритм Дамерау-Левенштейна. Исходя из замеров по памяти, итеративные алгоритмы проигрывают рекурсивным, потому что максимальный размер памяти в них растет, как произведение длин строк, а в рекурсивных — как сумма длин строк. Также при проведении эксперимента было выявлено, что на длине строк в 4 символа рекурсивная реализация алгоритма Дамерау-Левенштейна уже в 10 раз медленнее матричной реализации. При увеличении длины строк в геометрической прогрессии растет и время работы рекурсивной реализации. Следовательно, стоит использовать матричную реализацию для строк длиной более 4 символов.

Цель, которая была поставлена в начале лабораторной работы, была достигнута.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

[1] Вычисление редакционного расстояния [Электронный ресурс]. Режим доступа: https://habr.com/ru/articles/117063/ (дата обращения: 19.09.2023).

[2] Python [Электронный ресурс]. Режим доступа: https://www.python.org (дата обращения: 19.09.2023).

[3] time — Time access and conversions [Электронный ресурс]. Режим доступа: https://docs.python.org/3/library/time.html#functions (дата обращения: 19.09.2023).

[4] Ubuntu 22.04.3 LTS (Jammy Jellyfish) [Электронный ресурс]. Режим доступа: https://releases.ubuntu.com/22.04/ (дата обращения: 19.09.2023).

[5] Linux [Электронный ресурс]. Режим доступа: https://www.linux.org/ (дата обращения: 19.09.2023).