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»

Лабораторная работа № 1
по дисциплине «Анализ Алгоритмов»

Тема Расстояние Левенштейна

Студент Пермякова Е. Д.

Группа ИУ7-52Б

Преподаватели Строганов Д. В., Волкова Л. Л

Москва, 2024

Содержание

ВВЕДЕНИЕ

Целью работы является сравнительный анализ алгоритмов нахождения расстояний Левенштейна и Дамерау—Левенштейна

Задачи:

1) Изучить алгоритмы расстояни Левенштейна и Дамерау–Левенштейна;
2) Реализовать алгоритмы:
   — рекурсивный алгоритм поиска расстояния Левенштейна;
   — рекурсивный алгоритм поиска расстояния Левенштейна с кешированием;
   — нерекурсивный алгоритм поиска расстояния Левенштейна;
   — нерекурсивный алгоритм поиска расстояния Дамерау–Левенштейна;
3) Провести сравнительный анализ по времени работы алгоритмов

1 Аналитическая часть

В данной работе будут рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау–Левенштейна.

1.1 Расстояние Левенштейна

Расстояние Левенштейна — мера различия двух последовательностей символов (строк) относительно минимального количества операций вставки, удаления и замены, необходимых для перевода одной строки в другую [1].

Пусть S_1 и S_2 - две строки, длиной *M* и *N* соответственно, над некоторым алфавитом, тогда расстояние Левенштейна d(s_1, s_2) можно подсчитать по следующей рекуррентной формуле d(s_1, s_2) = D(M, N):

$$D(i,j) = \begin{cases} 0 & \text{i = 0, j = 0} \\ i & \text{j = 0, i > 0} \\ j & \text{i = 0, j > 0} \\ \min\{ & \\ \qquad D(i, j-1) + 1 & \\ \qquad D(i-1, j) + 1 & \text{i > 0, j > 0} \\ \qquad D(i-1, j-1) + m(s_1[i], s_1[j]) & (1.2) \\ \} & \end{cases} \tag{1.1}$$

Функция m(a, b) определена как:

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

1.2 Рекурсивный алгоритм нахождения расстояния Левенштейна

Рекурсивный алгоритм нахождения расстояния Левенштейна реализует формулу 1.1. Основная идея состоит в том, чтобы последовательно сравнивать символы двух строк, начиная с конца, и рекурсивно находить минимальное количество операций для преобразования предыдущих частей строк. Рекурсивный алгоритм неэффективен, так как многие вычисления производятся повторно.

1.3 Рекурсивный алгоритм нахождения расстояния Левенштейна с использованием кеша

Рекурсивный алгоритм нахождения расстояния Левенштейна с использованием кеша является оптимизацией предыдущего алгорита. Основная идея состоит в том, чтобы избежать повторных вычислений, сохраняя уже вычисленные промежуточные результаты и используя их вместо перерасчета.

1.4 Матричный алгоритм нахождения расстояния Левенштейна

Матричный алгоритм нахождения расстояния Левенштейна работает путём построения матрицы, в которую записывается минимальное расстояние Левенштейна для подстрок. Матрица имеет размеры (N+1) * (M+1), первая строка заполняется числами от 0 до M, а первый столбец – числами от 0 до N. Ячейки матрицы заполняются формуле:

$$matrix[i][j] = min \begin{cases} matrix[i-1][j] + 1 \\ matrix[i][j-1] + 1 \\ matrix[i-1][j-1] + m(s_1[i], s_2[j]) \quad (1.4) \end{cases} . \quad (1.3)$$

Функция 1.4 определена как:

$$m(s_1[i], s_2[j]) = \begin{cases} 0, & \text{если } s_1[i-1] = s_2[j-1], \\ 1, & \text{иначе} \end{cases} . \quad (1.4)$$

Результат вычисления расстояния Левенштейна будет в ячейке матрицы с индексами $i = N$ и $j = M$.

1.5 Алгоритм нахождения расстояния Дамерау–Левенштейна

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

Расстояние Дамерау–Левенштейна может быть вычислено по рекуррентной формуле:

$$D(i,j) = \begin{cases} 0, & \text{i = 0, j = 0,} \\ i, & \text{j = 0, i > 0,} \\ j, & \text{i = 0, j > 0,} \\ min \begin{cases} D(i,j-1)+1, \\ D(i-1,j)+1, \\ D(i-1,j-1)+m(S_1[i],S_2[j]) \quad (1.4) \\ , \\ D(i-2,j-2)+1, \end{cases} & \begin{array}{l} \text{если i > 1, j > 1,} \\ S_1[i] = S_2[j-1], \\ S_1[i-1] = S_2[j], \end{array} \\ min \begin{cases} D(i,j-1)+1, \\ D(i-1,j)+1, \\ D(i-1,j-1)+m(S_1[i],S_2[j]) \quad (1.4) \\ , \end{cases} & \text{иначе.} \end{cases} \tag{1.5}$$

Вывод

В данном разделе были теоретически разобраны рекуррентная формула расстояния Левенштейна и различные ее модификации и формула расстояния Дамерау–Левенштейна.

2 Конструкторская часть

В этом разделе будут представлены схемы алгоритмов вычисления расстояния Левенштейна и Дамерау–Левенштейн.

2.1 Описание алгоритмов

На рисунках 2.1-2.5 представлены схемы алгоритмов вычисления расстояния Левенштейна и Дамерау–Левенштейна.

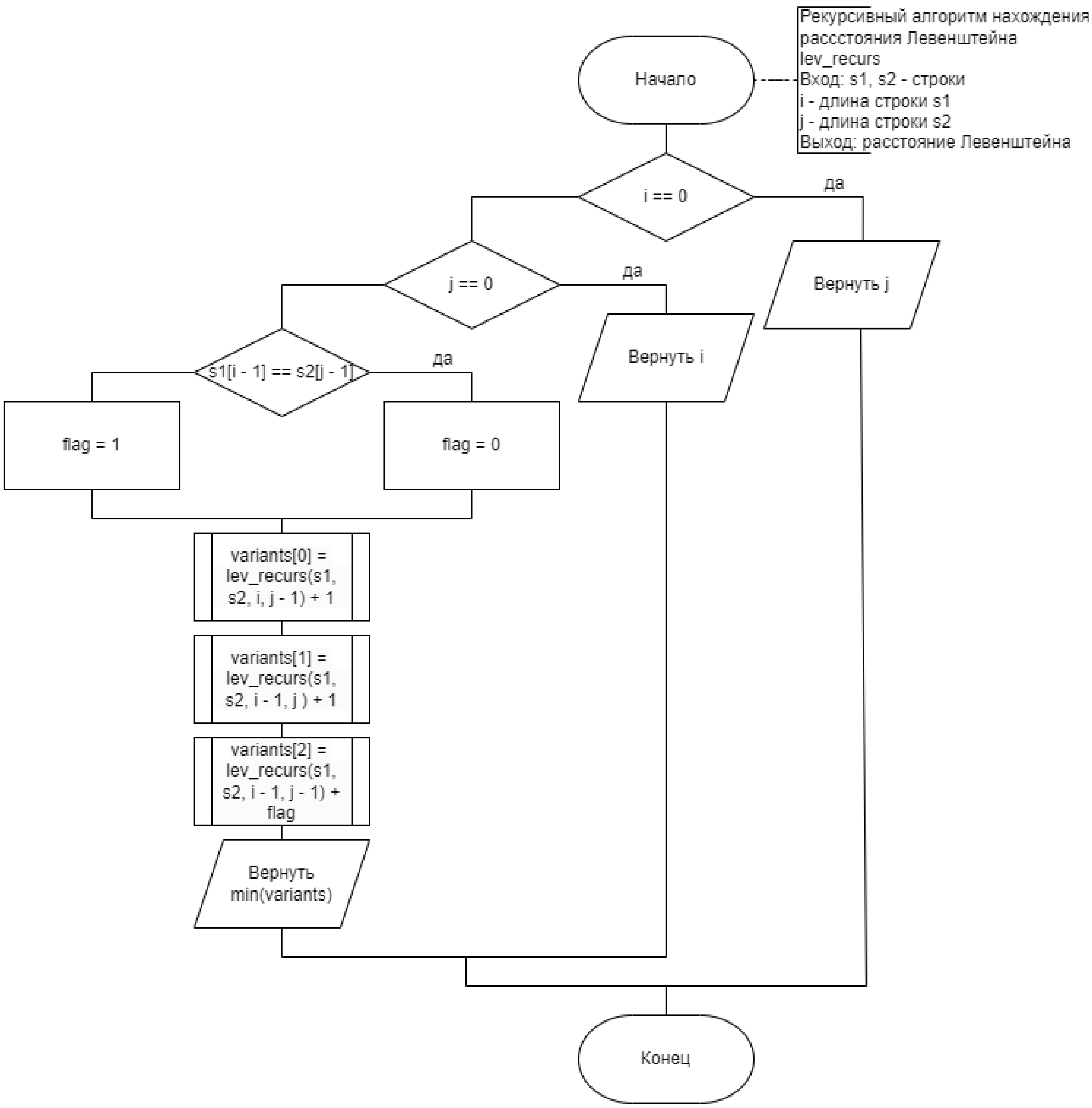

Рисунок 2.1 — Схема рекурсивного алгоритма нахождения расстояния Левенштейна

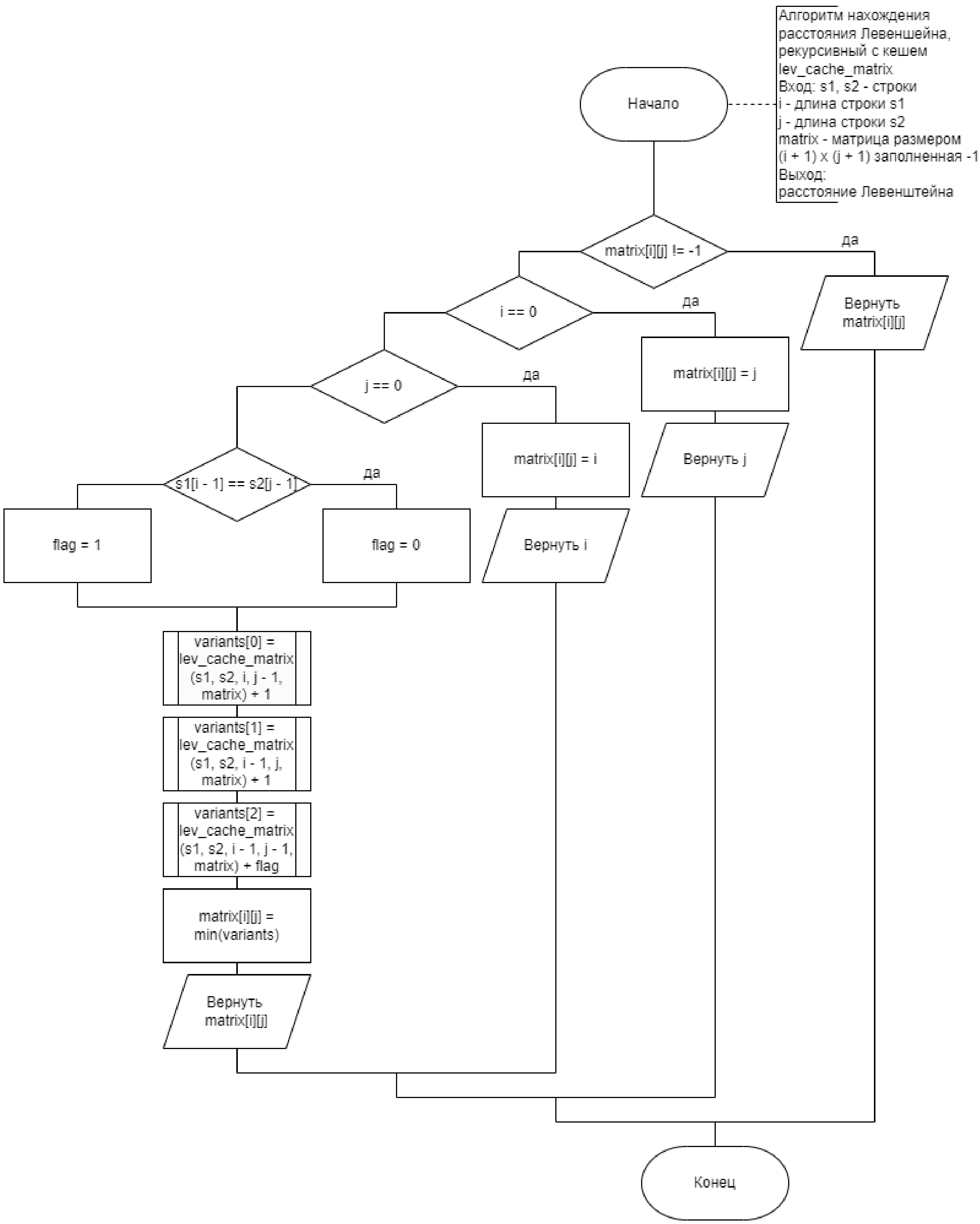

Рисунок 2.2 — Схема рекурсивного алгоритма нахождения расстояния Левенштейна с использованием кеша (матрицы)

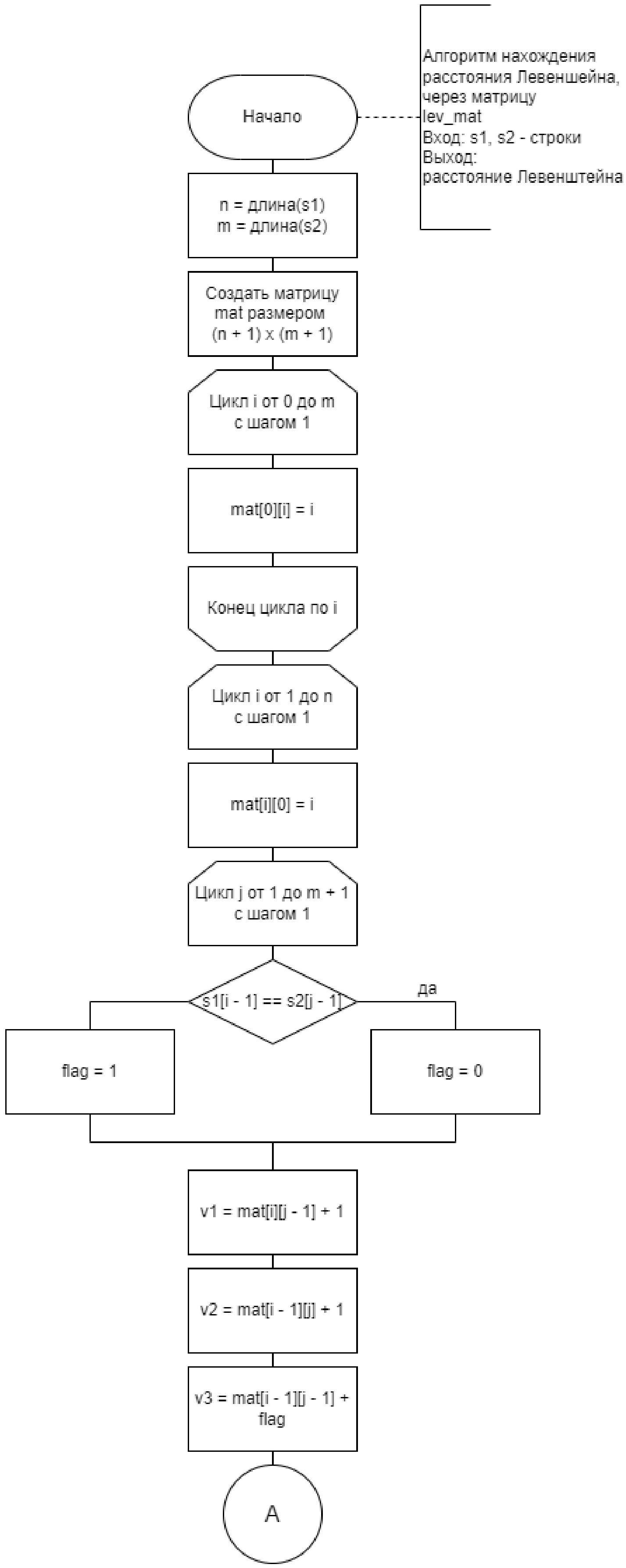

Рисунок 2.3 — Схема матричного алгоритма нахождения расстояния Левенштейна

Рисунок 2.4 — Схема матричного алгоритма нахождения расстояния Левенштейна

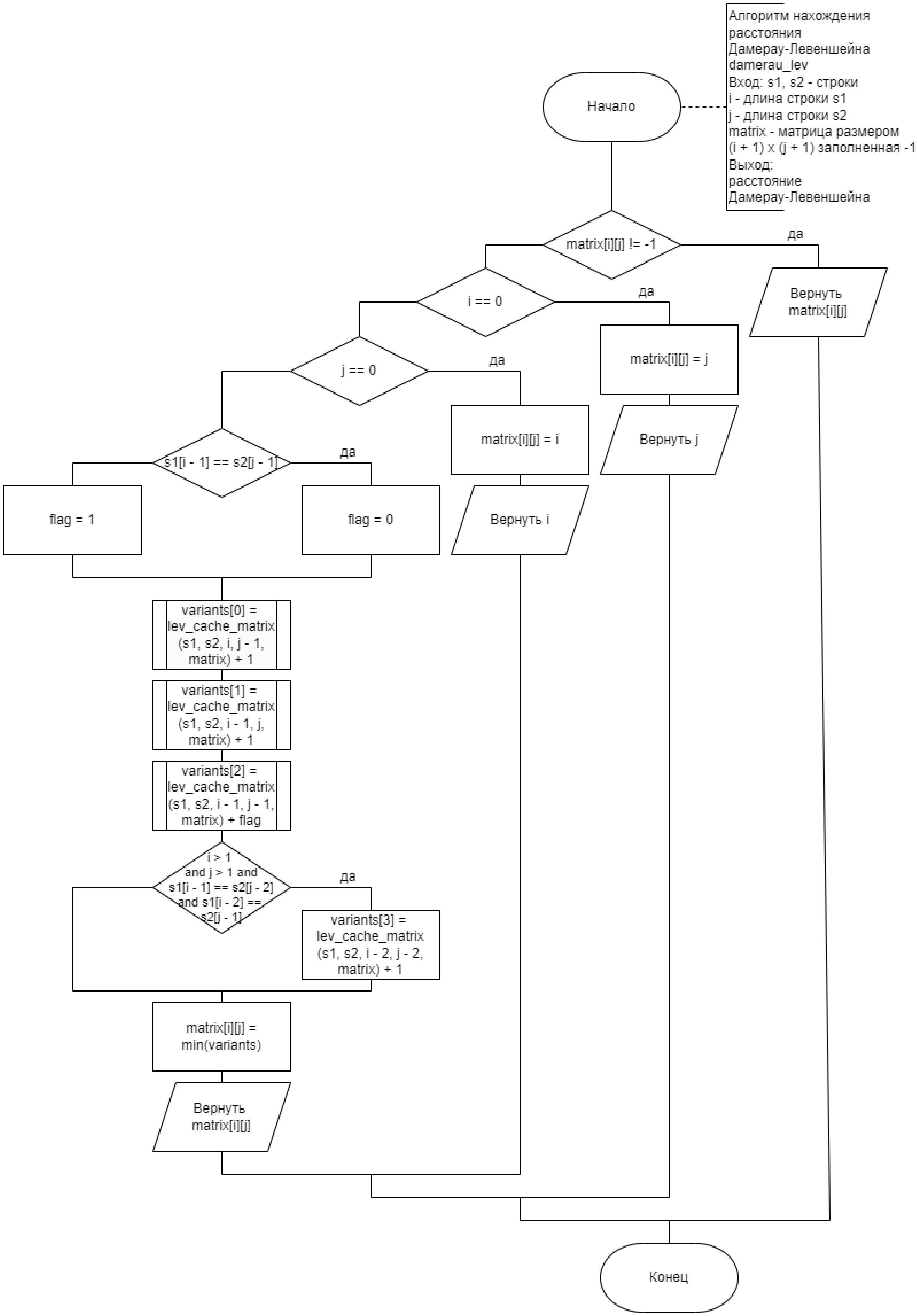

Рисунок 2.5 — Схема алгоритма нахождения расстояния Дамерау–Левенштейна

Вывод

В данном разделе были представлены схемы алгоритмов нахождения расстояний Левенштейна и Дамерау–Левенштейна.

3 Технологическая часть

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

3.1 Средства реализации

В данной работе для реализации был выбран язык программирования $Python$ [2], так как он удовлетворяет требованиям лабраторной работы: может замерить процессорное время (с помошью функции $process\_time\_ns(...)$ из библиотеки $time$ [3])

3.2 Реализация алгоритмов

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

Листинг 3.1 – Рекурсивный алгоритм нахождения расстояния Левенштейна

```
def lev_recurs_(s1, s2, i, j):
    if i == 0:
        return j
    if j == 0:
        return i

    flag = 0 if s1[i - 1] == s2[j - 1] else 1

    variants = [
        lev_recurs_(s1, s2, i, j - 1) + 1,
        lev_recurs_(s1, s2, i - 1, j) + 1,
        lev_recurs_(s1, s2, i - 1, j - 1) + flag
    ]
    return min(variants)


def lev_recurs(s1, s2):
    return lev_recurs_(s1, s2, len(s1), len(s2))
```

Листинг 3.2 – Рекурсивный алгоритм нахождения расстояния Левенштейна с использованием кеша

```
def lev_cache_matrix(s1, s2, i, j, matrix):
    if matrix[i][j] != -1:
        return matrix[i][j]

    if i == 0:
        matrix[i][j] = j
        return j
    if j == 0:
        matrix[i][j] = i
        return i

    flag = 0 if s1[i - 1] == s2[j - 1] else 1
    variants = [
        lev_cache_matrix(s1, s2, i, j - 1, matrix) + 1,
        lev_cache_matrix(s1, s2, i - 1, j, matrix) + 1,
        lev_cache_matrix(s1, s2, i - 1, j - 1, matrix) +
           flag,
    ]
    matrix[i][j] = min(variants)

    return matrix[i][j]


def lev_cache(s1, s2, printing=False):
    i, j = len(s1), len(s2)
    matrix = []
    for i in range(len(s1) + 1):
        matrix.append([-1] * (len(s2) + 1))

    res = lev_cache_matrix(s1, s2, i, j, matrix)
    if printing:
        for i in range(len(s1) + 1):
            print(matrix[i])
    return res
```

Листинг 3.3 – Матричный алгоритм нахождения расстояния Левенштейна

```
def lev_mat(s1, s2, printing=False):
    mat = [[i for i in range(len(s2) + 1)]]
    if printing: print(mat[0])
```

```
        for i in range(len(s1)):
            mat.append([i + 1])
            i += 1
            for j in range(1, len(s2) + 1):
                flag = 0 if s1[i - 1] == s2[j - 1] else 1
                new = min(
                    mat[i][j - 1] + 1,
                    mat[i - 1][j] + 1,
                    mat[i - 1][j - 1] + flag
                )

        mat[i].append(new)

        if printing: print(mat[i])
            return mat[-1][-1]
```

Листинг 3.4 – Алгоритм нахождения расстояния Дамерау–Левенштейна

```
        def damerau_lev_matrix(s1, s2, i, j, matrix):
        if matrix[i][j] != -1:
            return matrix[i][j]

        if i == 0:
            matrix[i][j] = j
            return j
        if j == 0:
            matrix[i][j] = i
            return i

        flag = 0 if s1[i - 1] == s2[j - 1] else 1
        variants = [
            damerau_lev_matrix(s1, s2, i, j - 1, matrix) + 1,
            damerau_lev_matrix(s1, s2, i - 1, j, matrix) + 1,
            damerau_lev_matrix(s1, s2, i - 1, j - 1, matrix) +
               flag,
        ]
        if i > 1 and j > 1 and s1[i - 1] == s2[j - 2] and s1[i
          - 2] == s2[j - 1]:
            variants.append(
                damerau_lev_matrix(s1, s2, i - 2, j - 2,
                   matrix) + 1
            )
```

```
        matrix[i][j] = min(variants)

        return matrix[i][j]


def damerau_lev(s1, s2, printing=False):
    i, j = len(s1), len(s2)
    matrix = []
    for i in range(len(s1) + 1):
        matrix.append([-1] * (len(s2) + 1))

    res = damerau_lev_matrix(s1, s2, i, j, matrix)
    if printing:
        for i in range(len(s1) + 1):
            print(matrix[i])
    return res
```

3.3 Классы эквивалентности тестирования

Для тестирования были выделены следующие классы тестирования:

1) Пустые строки
2) Одна строка пустая, другая нет
3) Строки равной длины
4) Анаграммы
5) Расстояния, которые вычислены алгоритмами Левенштейна и ДамерауЛевенштейна, равны
6) Расстояния, которые вычислены алгоритмами Левенштейна и ДамерауЛевенштейна, дают разные результаты

3.4 Функциональные тесты

В таблице 3.1 приведены функциональные тесты для алгоритмов нахождения расстояния Левенштейна и Дамерау–Левенштейна.

Все тесты пройдены успешно.

Таблица 3.1 — Функциональные тесты

			Результат	
№	Строка 1	Строка 2	Левенштейн	Дамерау–Л.
1	"Пустая строка"	"Пустая строка"	0	0
2	"Пустая строка"	Слово	5	5
3	Мука	"Пустая строка"	4	4
4	мука	река	2	2
5	абвг	гбав	3	3
6	123	123456	3	3
7	1234	2134	2	1

Вывод

Были представлены листинги всех описанных ренее алгоритмов нахождения расстояния Левенштейна и Дамерау–Левенштейна и их тесты.

4 Исследовательская часть

Цель исследования - определение зависимости времени работы алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна от размера поданых на вход строк.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- Операционная система – Майкрософт Windows 11 Домашняя для одного языка; Версия - 10.0.22631; Сборка - 22631;
- Установленная оперативная память (RAM) - 16,0 ГБ;
- Процессор - AMD Ryzen 7 5800H with Radeon Graphics, 3201 МГц, ядер: 8, логических процессоров: 16;

При тестировании ноутбук был включен в сеть электропитания, Во время тестирования на ноутбуке были запущены только встроенное приложение окружения PyCharm и система тестирования.

4.2 Время выполнения алгоритмов

Как было сказано выше для замера процессорного времени использовалась функция process_time_ns(...) из библиотеки time на Python.

Для графика 4.1 замеры проводились для длины слов от 0 до 12. А для графика 4.2 для длины слов от 0 до 500, так как время работы рекурсивного алгоритма возрастает, намного быстрее, чем остальных.

Замеры времени проводились по принципу: для одних входных данные проводилось 10 замеров и если относительная стандартная ошибка среднего (rse) была >= 5%, то для этих данных замеры продолжались, в таблицу заносилось среднее значение.

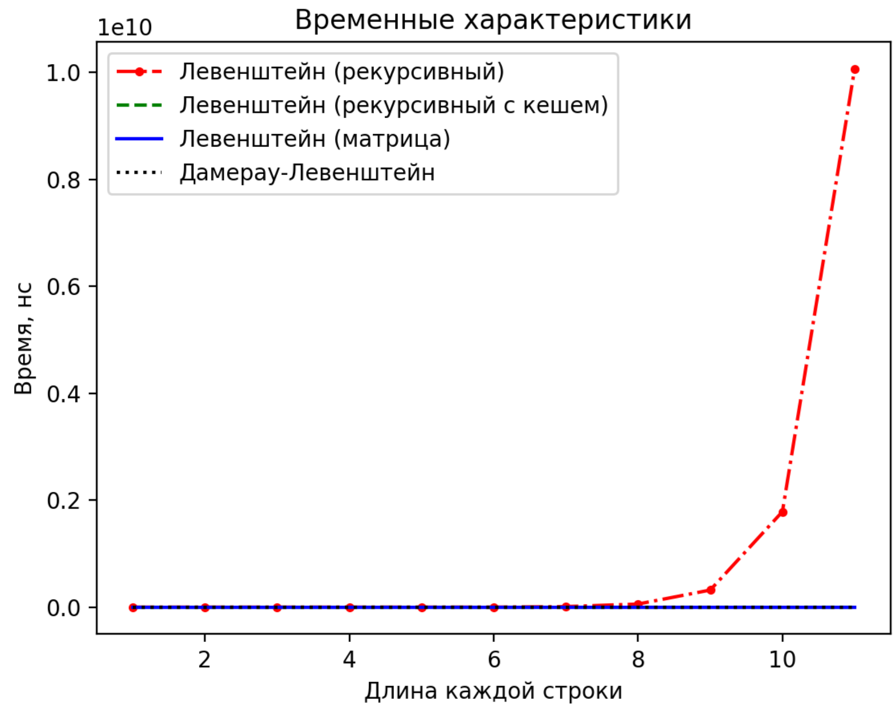

Рисунок 4.1 — Сравнение по времени алгоритмов Левенштейна и Дамерау–Левенштейна

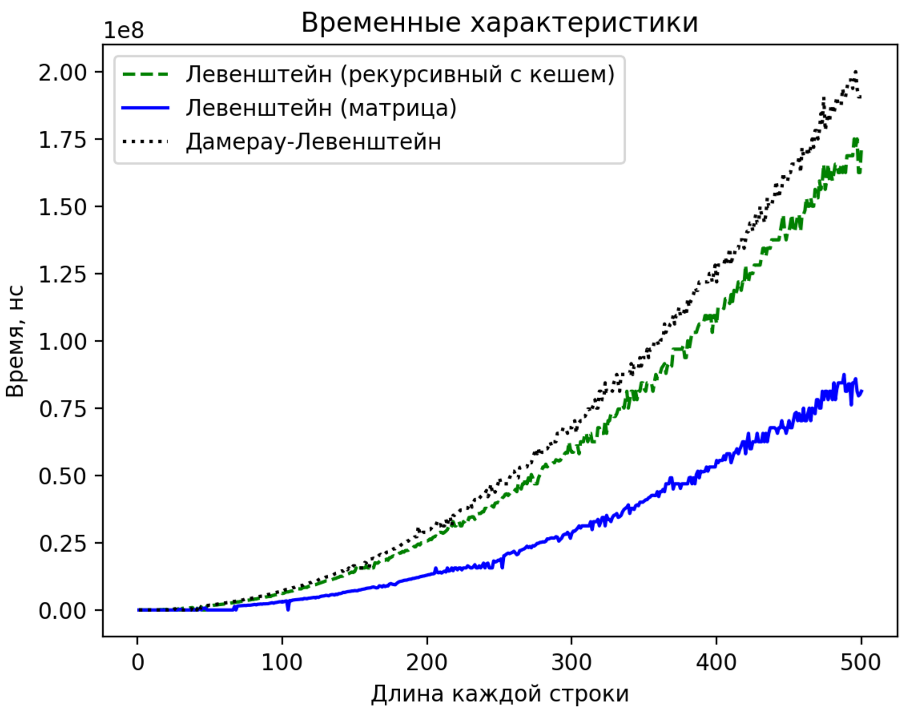

Рисунок 4.2 — Сравнение по времени алгоритмов Левенштейна с использованием кеша, матричной реализации и Дамерау–Левенштейна

4.3 Вывод

Из проведённых замеров можно сделать следующие выводы:

— Матричная реализация алгоритма Левенштейна демонстрирует наилучшие результаты по времени работы на всех тестовых данных.

— Рекурсивная реализация алгоритма Левенштейна без мемоизации значительно проигрывает по времени всем другим подходам.

ЗАКЛЮЧЕНИЕ

В ходе лабораторной работы была выполнена поставленнная цель, которая заключалась в выполнении сравнительного анализа алгоритмов нахождения расстояний Левенштейна и Дамерау—Левенштейна. Были выполнены следующие задачи:

1) были рассмотрены алгоритмы Левенштейна и
Дамерау–Левенштейна нахождения расстояния между строками;
2) были реализованы следующие алгоритмы;
    — Рекурсивный алгоритм нахождения расстояния Левенштейна;
    — Рекурсивный алгоритм нахождения расстояния Левенштейна с использованием кеша;
    — Матричный алгоритм нахождения расстояния Левенштейна;
    — Алгоритм нахождения расстояния Дамерау–Левенштейна;
3) был проведен сравнительный анализ алгоритмов по времени;

Основываясь на проведенном исследовании можно сделать следующие выводы:

— Матричная реализация алгоритма Левенштейна демонстрирует наилучшие результаты по времени работы на всех тестовых данных.
— Рекурсивная реализация алгоритма Левенштейна без мемоизации значительно проигрывает по времени всем другим подходам.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Карахтанов Д. С. Программная реализация алгоритма Левенштейна для устранения опечаток в записях баз данных // Молодой ученый. 2010. Т. Т. 1, № № 8 (19). С. 158–162. (дата обращения: 23.09.2024). URL: https://moluch.ru/archive/19/1966/.

2. Welcome to Python [Электронный ресурс]. Режим доступа: `https://www.python.org`. (дата обращения: 23.09.2024).

3. time — Time access and conversions [Электронный ресурс]. Режим доступа: `https://docs.python.org/3/library/time.html#functions`. (дата обращения: 23.09.2024).