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 Кузнецова А. В.

Группа ИУ7-51Б

Оценка (баллы) ______

Преподаватель Волкова Л. Л.

Москва — 2022 г.

Оглавление

Введение

Расстояние Левенштейна — это минимальное количество редакторских операций, необходимых для преобразования одной строки в другую. Редакторские операции:

- вставка 1 символа;
- удаление 1 символа;
- замена 1 символа.

Преобразовать одно слово в другое можно различными способами, количество действий также может быть разным. При вычислении расстояния Левенштейна следует выбирать минимальное количество действий.

Исследования Ф. Дамерау показали, что наиболее частая ошибка при наборе слова — случайная перестановка двух соседних букв. В случае одной транспозиции расстояние Левенштейна равно 2. При использовании поправки Дамерау транспозиция принимается за единичное расстояние [1]. Для расстояния Дамерау-Левенштейна к редакторским операциям, указанным выше добавляется транспозиция — перестановка двух соседних символов.

Алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна находят применение в следующих областях:

- компьютерная лингвистика (задача автозамены и исправления ошибок);
- биоинформатика (задача анализа иммунитета).

Целью данной лабораторной работы является изучение методов динамического программирования на материале расстояний Левенштейна и Дамерау-Левенштейна. Для достижения поставленной цели требуется решить следующие задачи:

1. Изучение расстояний Левенштейна и Дамерау-Левенштейна;
2. Создание схем изучаемых алгоритмов;

3. Реализация алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна;
4. Оценка затрат алгоритмов по памяти;
5. Определение средств программной реализации выбранных алгоритмов;
6. Выполнение замеров процессорного времени работы реализаций алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна;
7. Проведение сравнительного анализа по времени нерекурсивных алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна;
8. Проведение сравнительного анализа по времени трёх алгоритмов поиска расстояний Дамерау-Левенштейна;
9. Подготовка отчета о выполненной лабораторной работе.

1 Аналитическая часть

В данном разделе будет представлено теоретическое описание алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.

1.1 Матричный алгоритм поиска расстояния Левенштейна

Рассмотрим перевод строки $A' = A[:\ i]$ в строку $B' = B[:\ j]$, последние символы которых соответственно a_{i-1} и b_{j-1}, при помощи следующих операций, каждая из которых имеет свою стоимость (x, y - символы, λ - строка):

- вставка символа в произвольное место - I, стоимостью $w(x, \lambda) = 1$;
- удаление символа с произвольной позиции - D, $w(\lambda, y) = 1$;
- замена символа на другой - R, $w(x, y) = 1$, $x \neq y$.

Расстояние Левенштейна — минимальное количество операций I, D, R для перевода $A' = A[:\ i]$ в $B' = B[:\ j]$ [1]. Его можно подсчитать по следующей реккурентной формуле:

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

Функция m определена как:

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

Прямая реализация формулы 1.1 может быть малоэффективна по времени исполнения при больших i, j, т. к. множество промежуточных значений D(i, j) вычисляются заново множество раз подряд. Для оптимизации нахождения расстояния Левенштейна можно использовать матрицу в целях хранения соответствующих промежуточных значений (выполнять меморизацию промежуточных значений).

Матрицу M размером $N+1$ и $L+1$, где N и L — длины строк $A' = A[: i]$ и $B' = B[: j]$, заполняют по следующей формуле:

$$M[i][j] = \begin{cases} 0 & \text{i = 0, j = 0} \\ i & \text{j = 0, i > 0} \\ j & \text{i = 0, j > 0} \\ \min\{ & \\ \qquad M[i][j-1] + 1 & \\ \qquad M[i-1][j] + 1 & \\ \qquad M[i-1][j-1] + m(A[: i-1], B[: j-1]) & \\ \} & \end{cases} \tag{1.3}$$

Функция m определена как:

$$m(x, y) = \begin{cases} 0 & \text{если x = y} \\ 1 & \text{иначе} \end{cases} \tag{1.4}$$

$M[N][L]$ содержит значение расстояния Левенштейна.

1.2 Рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна

Как было сказано раннее, алгоритм Дамерау-Левенштейна отличается от алгоритма Левенштейна добавлением операции транспозиции символов при переводе одной строки в другую. Расстояние Дамерау-Левенштейна может быть найдено по формуле:

$$d(i,j) = \begin{cases} \max(i,j), & \text{если } \min(i,j) = 0, \\ \min\{ & \\ \qquad d(i,j-1)+1, & \\ \qquad d(i-1,j)+1, & \\ \qquad d(i-1,j-1)+m(a_{i-1},b_{j-1}), & \text{иначе} \\ \qquad \left[\begin{array}{ll} d(i-2,j-2)+1, & \text{если } i,j>1; \\ & a_i = b_{j-1}; \\ & b_j = a_{i-1} \\ \infty, & \text{иначе} \end{array}\right. & \\ \} & \end{cases} \tag{1.5}$$

Пусть $d(i,j)$ — есть расстояние Дамерау-Левенштейна между строками $A' = A[:i]$ и $B' = B[:j]$. $B' = B[:j]$ может быть получен из $A' = A[:i]$ следующими способами:

- если символ a_{i-1} был удален при редактировании, тогда необходимо $A[:i-1]$ превратить в $B[:j]$, на что необходимо $d(i-1,j)$ операций редактирования. В этом случае $d(i,j) = d(i-1,j)+1$;

- если символ a_{j-1} был добавлен при редактировании, то необходимо $A[:i]$ превратить в $B[:j-1]$, на что необходимо $d(i,j-1)$ операций редактирования. В этом случае $d(i,j) = d(i,j-1)+1$;

- если последние символы префиксов совпадают, т. е. $a_{i-1} = b_{j-1}$, то в этом случае можно не менять эти последние символы. Тогда $d(i, j) = d(i-1, j-1)$;

- если $a_{i-1} \neq b_{j-1}$, то тогда можно потратить 1 операцию на замену символа a_{i-1} на b_{j-1} и также потратить операцию $D(i-1, j-1)$ на превращение $A[: i-1]$ в $B[: j-1]$. Тогда $d(i, j) = d(i-1, j-1) + 1$;

- если последние два символа a_i и b_j были переставлены, то $d(i, j)$ может быть равно $d(i-2, j-2) + 1$.

Далее при вычислении необходимо взять минимум из всех перечисленных возможностей (при этом из случаев 3 или 4 рассматривается только один, в зависимости от условия $a_{i-1} = b_{j-1}$).

Начальные значения: $D(i, 0) = i$, $D(0, j) = j$.

1.3 Матричный алгоритм поиска расстояния Дамерау-Левенштейна

Прямая реализация формулы 1.5 так же может быть малоэффективна по времени исполнения при больших i, j, т. к. множество промежуточных значений D(i, j) вычисляются заново множество раз подряд. Для оптимизации можно использовать матрицу в целях хранения соответствующих промежуточных значений.

Матрицу M размером $N+1$ и $L+1$, где N и L — длины строк $A' = A[: i]$ и $B' = B[: j]$, заполняют по следующей формуле:

$$M[i][j] = \begin{cases} 0 & \text{i = 0, j = 0} \\ i & \text{j = 0, i > 0} \\ j & \text{i = 0, j > 0} \\ \min \begin{cases} M[i][j-1]+1 \\ M[i-1][j]+1 \\ M[i-1][j-1]+m(A[:i-1],B[:j-1]) \\ \left[\begin{array}{ll} M[i-2][j-2]+1, & \text{если } i,j>1; \\ & a_{i-1}=b_{j-2}; \\ & b_{j-1}=a_{i-2} \\ \infty, & \text{иначе} \end{array} \right. \end{cases} \end{cases} \tag{1.6}$$

$M[N][L]$ содержит значение расстояния Левенштейна.

1.4 Рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна с кешем

Проблему повторяющихся вычислений можно решить и при использовании рекурсии. Для этого достаточно создать кеш в виде матрицы, где будут храниться уже вычисленные значения. Если при выполении рекурсии происходит вызов с теми данными, которые ещё не были обработаны, то необходимое значение вычисляется и заносится в соответствующую ячейку матрицы. Если же данные уже были обработаны в дальнейших вычислениях участвует значение из матрицы. Таким образом, повторных вычислений не происходит.

Вывод

В данном разделе были описаны алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна. Алгоритмы могут быть реализованы рекурсивно или итерационно.

2 Конструкторская часть

В данном разделе будут рассмотрены схемы вышеизложенных алгоритмов.

2.1 Разработка алгоритмов

На рисунке 2.1 представлен матричный алгоритм поиска расстояния Левенштейна.

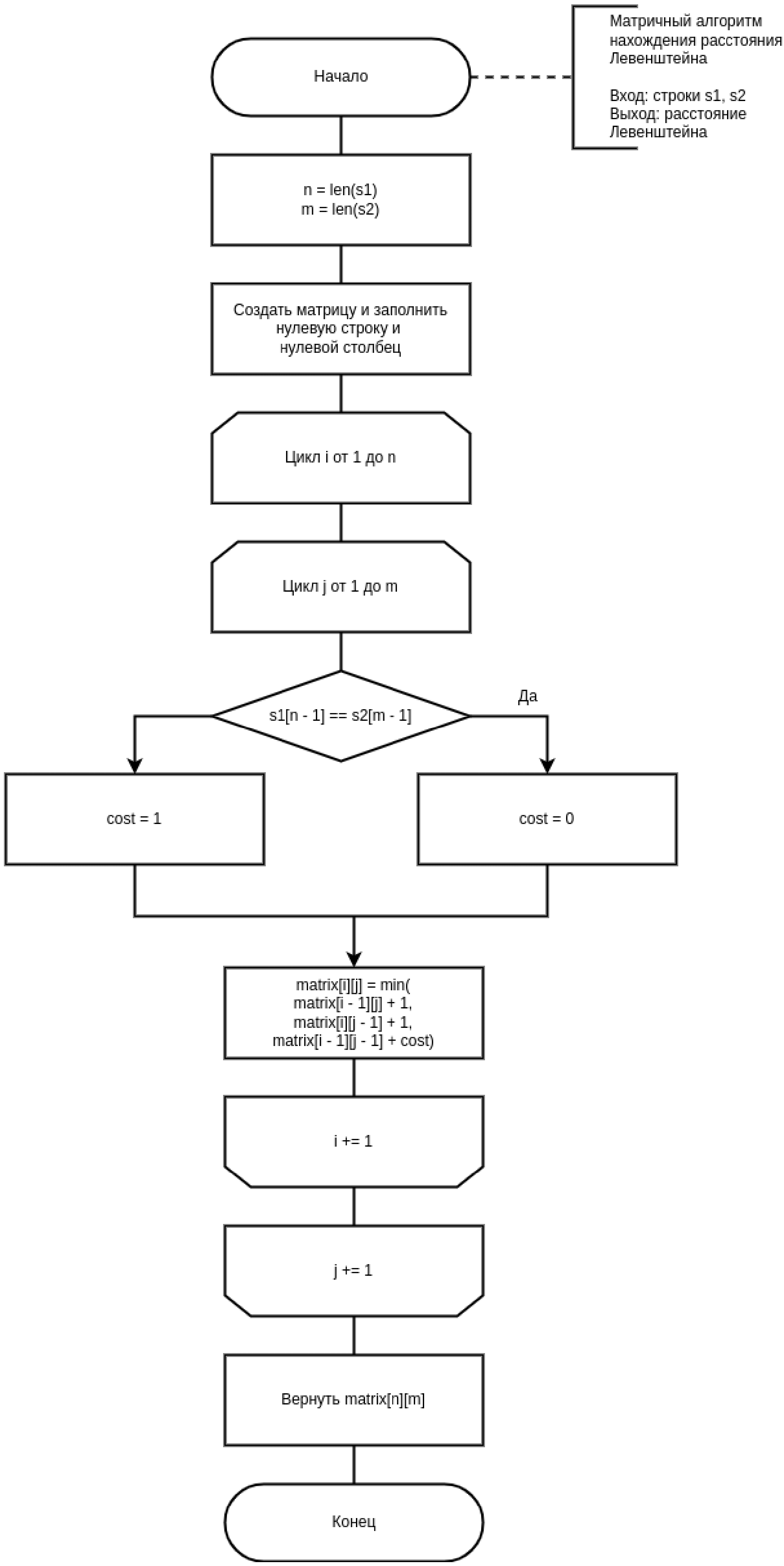

Рисунок 2.1 – Схема матричного алгоритма нахождения расстояния Левенштейна

На рисунке 2.2 представлен матричный алгоритм поиска расстояния Дамерау-Левенштейна.

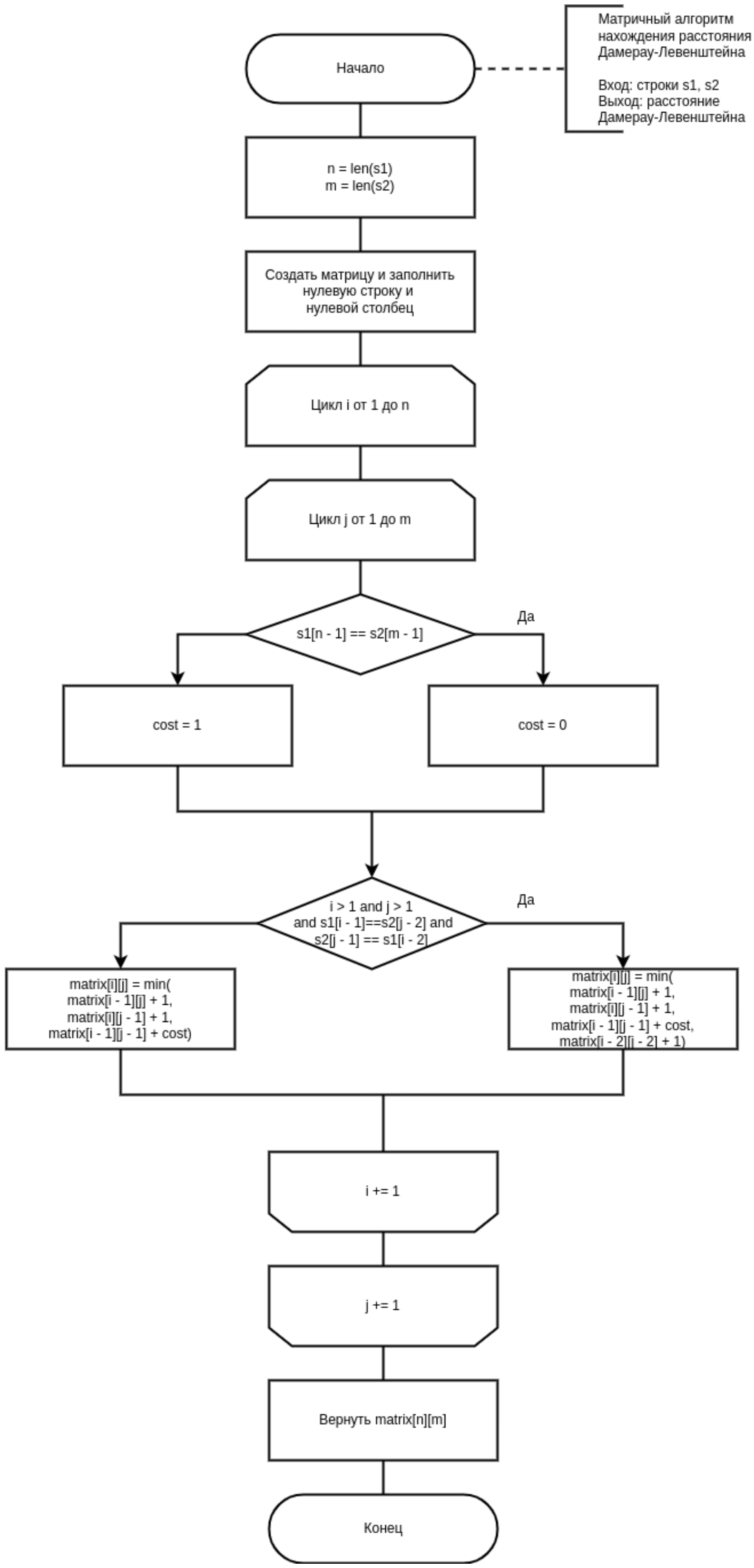

Рисунок 2.2 – Схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна

На рисунке 2.3 представлен рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна.

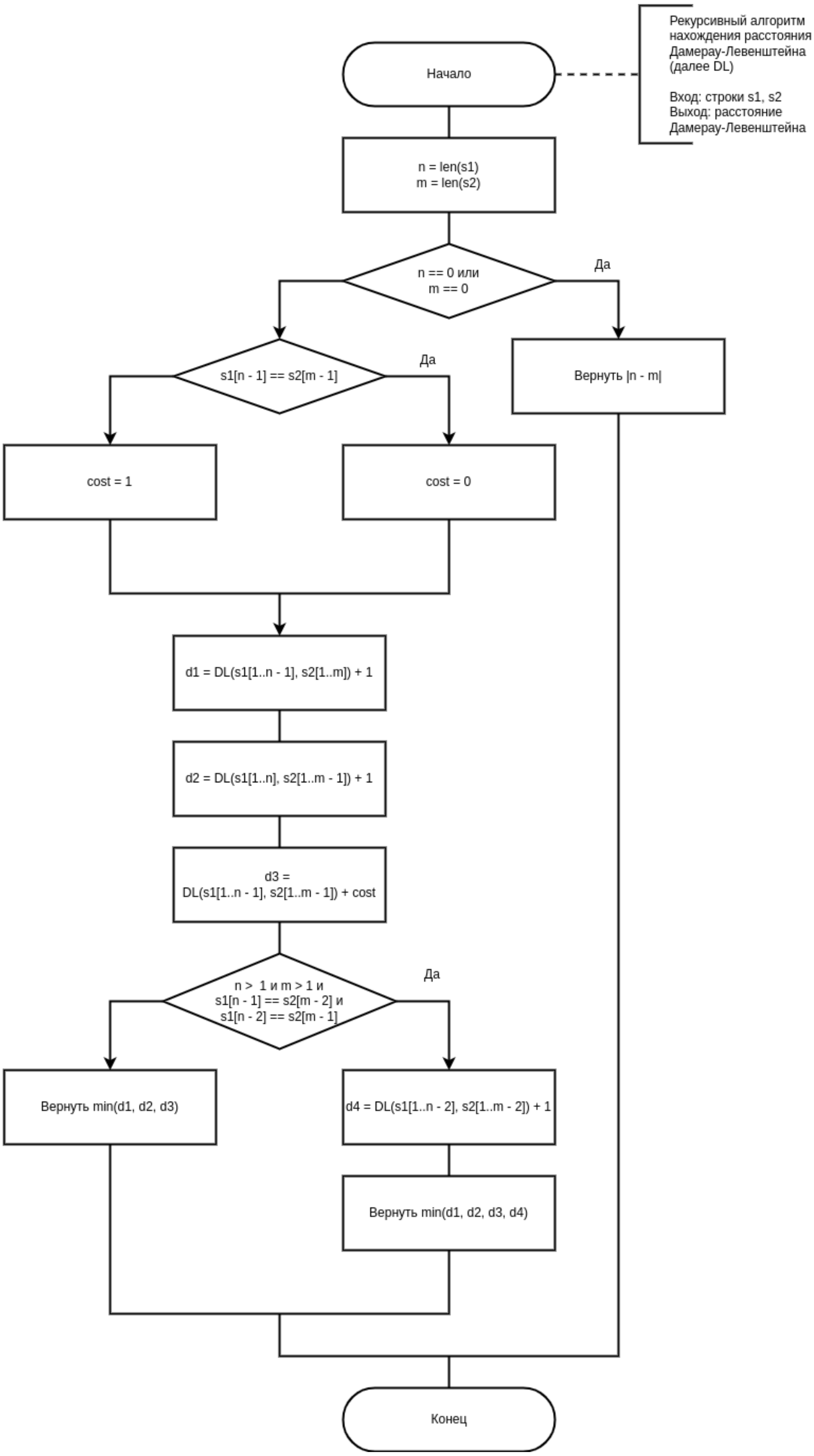

Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

На рисунке 2.4 представлен рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна с кешем.

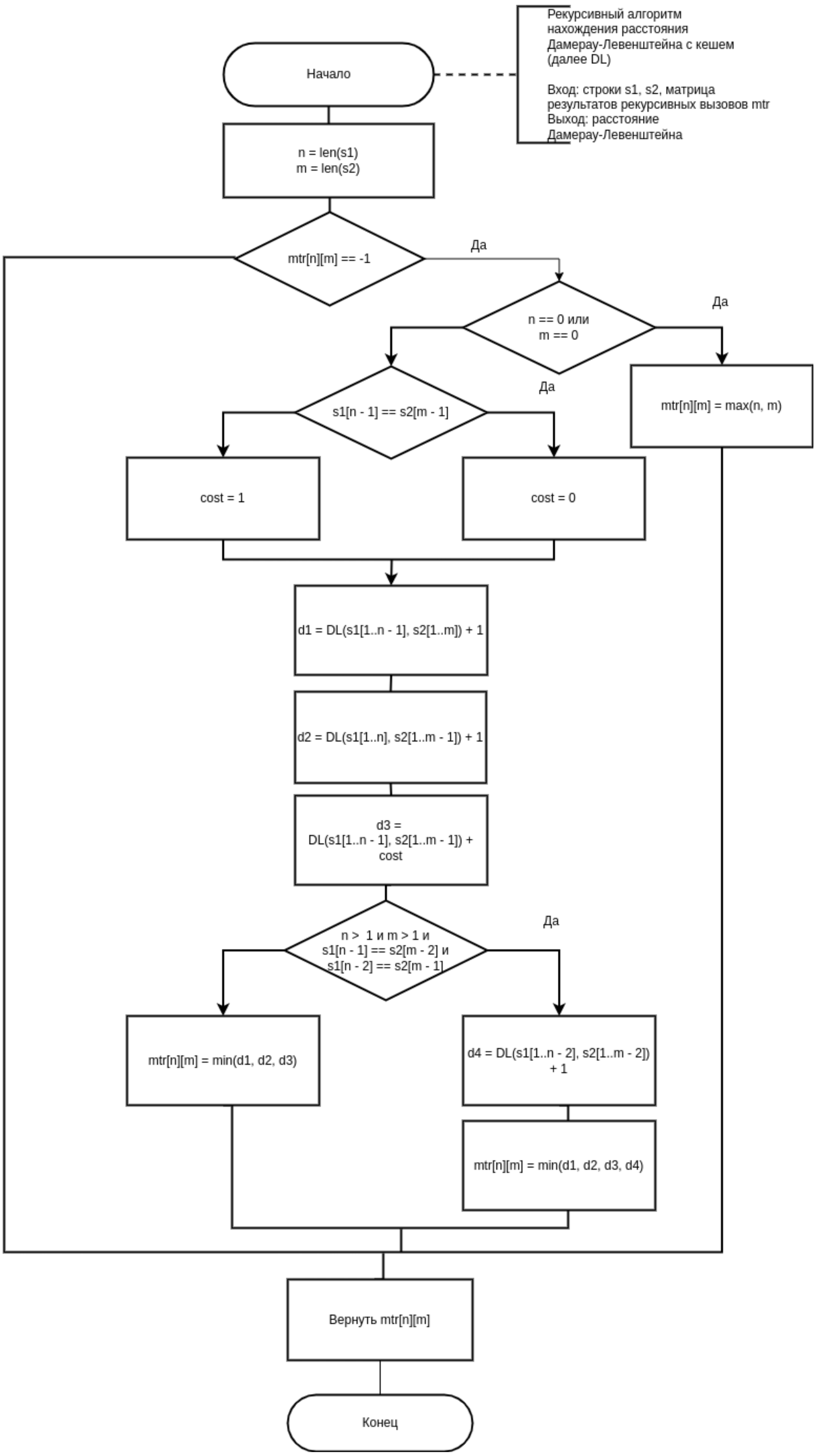

Рисунок 2.4 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с кешем

Вывод

На основе теоретических данных, полученных из аналитического раздела были построены схемы требуемых алгоритмов.

3 Технологическая часть

В данном разделе приведены средства реализации, сведения о модулях программы, листинги кода, тесты.

3.1 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язык Python [2]. Данный выбор обусловлен тем, что он позволяет реализовывать сложные задачи за короткие сроки за счет наличия большого количества подключаемых библиотек и простоты синтаксиса.

Замеры времени проводились при помощи функции process_time_ns из библиотеки time [3].

3.2 Сведения о модулях программы

Данная программа разбита на следующие модули:

- main.py - файл, содержащий точку входа в программу, в нем происходит вызов алгоритмов;
- levenshtein.py - файл, содержащий алгоритмы нахождения редакционного расстояния Левенштейна;
- damerau_levenshtein.py - файл, содержащий алгоритмы нахождения редакционного расстояния Дамерау-Левенштейна;
- matrix.py - файл, содержащий функции работы с матрицами;
- measurements.py - файл, содержащий функции замеров времени работы алгоритмов;

3.3 Листинги кода

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

Листинг 3.1 – Матричный алгоритм нахождения расстояния Левенштейна

```
def matrix_levenshtein(str1, str2):
    n1 = len(str1) + 1
    n2 = len(str2) + 1

    matrix = create_matrix(n1, n2)

    for i in range(1, n1):
        for j in range(1, n2):
            d1 = matrix[i][j - 1] + 1
            d2 = matrix[i - 1][j] + 1
            d3 = matrix[i - 1][j - 1] + (0 if str1[i - 1] ==
               str2[j - 1] else 1)
            matrix[i][j] = min(d1, d2, d3)

    return matrix[n1 - 1][n2 - 1]
```

Листинг 3.2 – Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна

```
def recursive_damerau_levenshtein(str1, str2):
    n1 = len(str1)
    n2 = len(str2)

    if n1 == 0 or n2 == 0:
        return max(n1, n2)

    d1 = recursive_damerau_levenshtein(str1[:], str2[:-1]) + 1
    d2 = recursive_damerau_levenshtein(str1[:-1], str2[:]) + 1
    d3 = recursive_damerau_levenshtein(str1[:-1], str2[:-1]) + (0
       if str1[-1] == str2[-1] else 1)

    if n1 > 1 and n2 > 1 and str1[-1] == str2[-2] and str2[-1] ==
       str1[-2]:
        d4 = recursive_damerau_levenshtein(str1[:-2], str2[:-2])
           + 1
        return min(d1, d2, d3, d4)
    else:
        return min(d1, d2, d3)
```

Листинг 3.3 – Матричный алгоритм нахождения расстояния Дамерау-Левенштейна

```
def matrix_damerau_levenshtein(str1, str2):
    n1 = len(str1) + 1
    n2 = len(str2) + 1

    matrix = create_matrix(n1, n2)

    for i in range(1, n1):
        for j in range(1, n2):
            d1 = matrix[i][j - 1] + 1
            d2 = matrix[i - 1][j] + 1
            d3 = matrix[i - 1][j - 1] + (0 if str1[i - 1] ==
                str2[j - 1] else 1)
            if i > 1 and j > 1 and str1[i - 1] == str2[j - 2] and
                str2[j - 1] == str1[i - 2]:
                d4 = matrix[i - 2][j - 2] + 1
                matrix[i][j] = min(d1, d2, d3, d4)
            else:
                matrix[i][j] = min(d1, d2, d3)

    return matrix[n1 - 1][n2 - 1]
```

Листинг 3.4 – Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна с кешем

```
def recursive_damerau_levenshtein_cache(str1, str2, matrix):
    n1 = len(str1)
    n2 = len(str2)

    if matrix[n1][n2] == -1:
        if n1 == 0 or n2 == 0:
            matrix[n1][n2] = max(n1, n2)

        else:
            d1 = rec_dam_lev_mtrx(str1[:], str2[:-1], matrix) + 1
            d2 = rec_dam_lev_mtrx(str1[:-1], str2[:], matrix) + 1
            d3 = rec_dam_lev_mtrx(str1[:-1], str2[:-1], matrix) +
            (0 if str1[-1] == str2[-1] else 1)

            if n1 > 1 and n2 > 1 and str1[-1] == str2[-2] and str2[-1] == str1[-2]:
                d4 = rec_dam_lev_mtrx(str1[:-2], str2[:-2], matrix) + 1
                matrix[n1][n2] = min(d1, d2, d3, d4)
            else:
                matrix[n1][n2] = min(d1, d2, d3)

    return matrix[n1][n2]
```

3.4 Тестирование

В таблице 3.1 приведены функциональные тесты для алгоритмов вычисления расстояния Левенштейна и Дамерау-Левенштейна.

Таблица 3.1 – Тесты

	Входные данные		Ожидаемый результат	
№	Строка 1	Строка 2	Левенштейн	Дамерау-Левенштейн
1	пустая строка	пустая строка	0	0
2	пустая строка	сон	3	3
3	корабль	пустая строка	7	7
4	мост	мос	1	1
5	кот	коооот	3	3
6	ура	уар	2	1
7	абв	ваб	2	2
8	повод	пвд	2	2

При проведении функционального тестирования, полученные результаты работы программы совпали с ожидаемыми. Таким образом, функциональное тестирование пройдено успешно.

Вывод

Были реализованы алгоритмы: вычисления расстояния Дамерау-Левенштейна с помощью матрицы, рекурсивно, рекурсивно с кешем, а также вычисления расстояния Левенштейна с помощью матрицы.

4 Исследовательская часть

В данном разделе будут приведены примеры работы программы, и будет проведен сравнительный анализ реализованных алгоритмов поиска редакционного расстояния по затраченному процессорному времени, а также анализ затрат по памяти.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось исследование приведены ниже.

- операционная система: Manjaro Linux [4];
- память : 7,6 GiB;
- процессор: 8 × Intel® Core™ i5-10210U CPU @ 1.60GHz [5].

4.2 Демонстрация работы программы

На рисунке 4.1 приведен пример работы программы.

```
МЕНЮ:
1 - Вычисление расстояния Левенштейна при помощи нерекурсивного алгоритма
2 - Вычисление расстояния Дамерау-Левенштейна при помощи нерекурсивного алгоритма
3 - Вычисление расстояния Дамерау-Левенштейна при помощи рекурсивного алгоритма
4 - Вычисление расстояния Дамерау-Левенштейна при помощи рекурсивного алгоритма с кешем
5 - Замеры времени
0 - Выход
Выбор:
4
Введите первую строку: даша
Введите вторую строку: шла

    0   ш   л   а
0   0   1   2   3
д   1   1   2   3
а   2   2   2   2
ш   3   2   3   3
а   4   3   3   3
Полученное расстояние: 3
```

Рисунок 4.1 – Пример работы программы

4.3 Временные характеристики

Функция process_time_ns из библиотеки time языка программирования Python возвращает процессорное время в наносекундах.

Замеры проводились для длины слов от 1 до 9 на случайных входных строках.

На рисунках 4.2-4.5 приведены графические результаты сравнения временных характеристик.

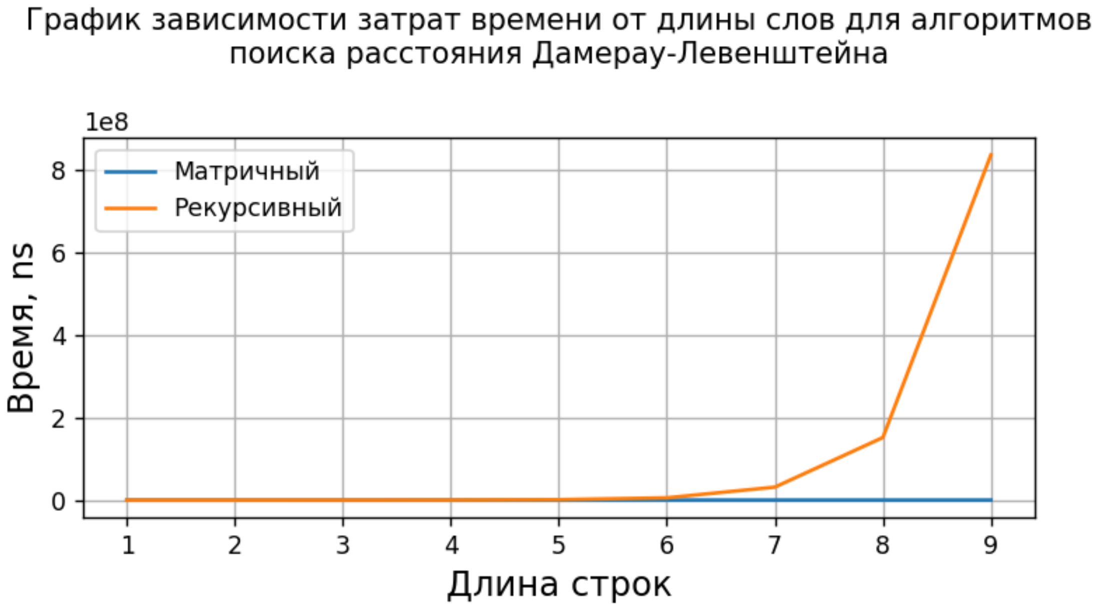

Рисунок 4.2 – Сравнение по времени рекурсивного алгоритма Дамерау-Левенштейна и матричного алгоритма Дамерау-Левенштейна

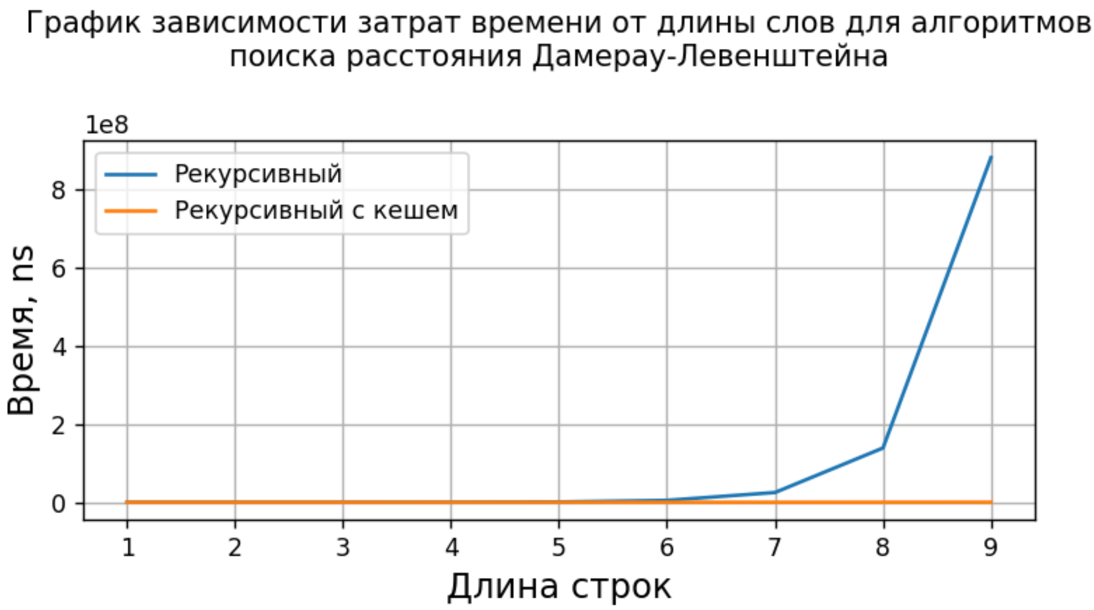

Рисунок 4.3 – Сравнение по времени рекурсивного алгоритма Дамерау-Левенштейна и рекурсивного алгоритма Дамерау-Левенштейна с кешем

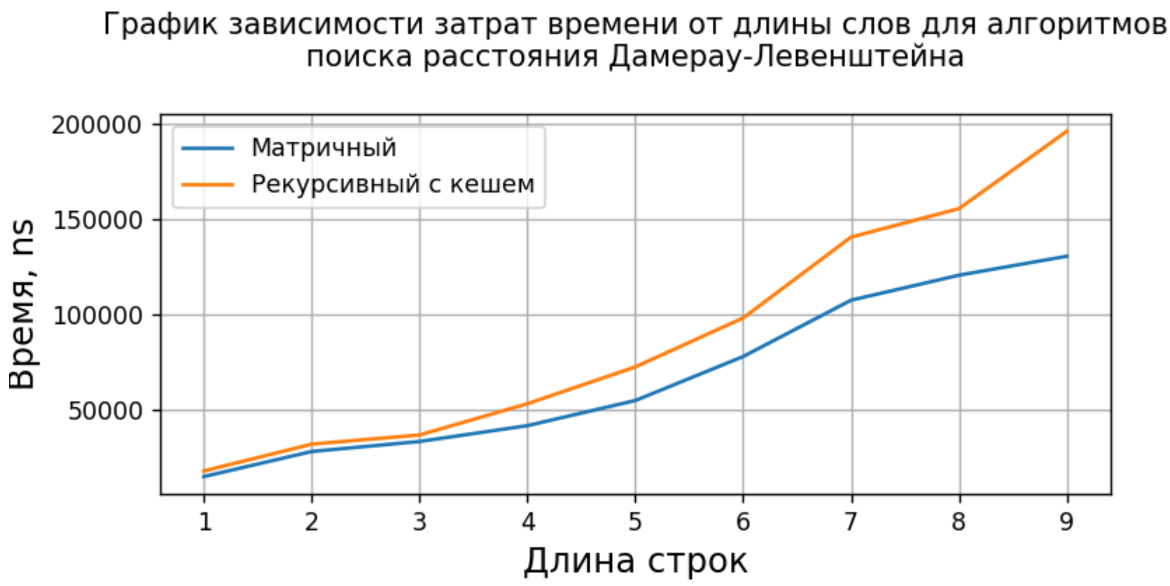

Рисунок 4.4 – Сравнение по времени рекурсивного алгоритма Дамерау-Левенштейна с кешем и матричного алгоритма Дамерау-Левенштейна

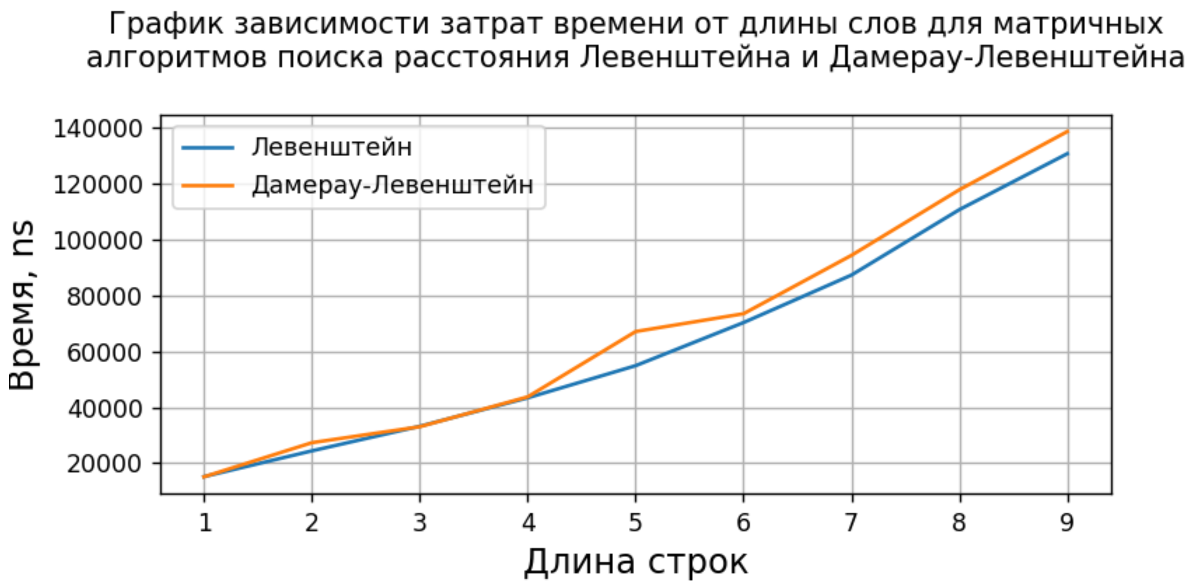

Рисунок 4.5 – Сравнение по времени матричного алгоритма Левенштейна и матричного алгоритма Дамерау-Левенштейна

4.4 Анализ затрат по памяти

Проведем анализ используемой памяти для рекурсивного, матричного и рекурсивного с кешем алгоритмов Дамерау-Левенштейна.

Для рекурсивного алгоритма при каждом вызове происходит выделение памяти под следующие типы данных:

- 2 строки типа str;
- 2 длины строк типа int;
- адрес возврата - m байт;
- 3 или 4 локальных переменных типа int.

При этом глубина рекурсии равна сумме длин двух строк. Таким образом, используемая память рекурсивного алгоритма в среднем равна:

$$M_r = (5,5 \cdot sizeof(int) + 2 \cdot sizeof(str) + m) \cdot (|s1| + |s2|) \tag{4.1}$$

Для нерекурсивного алгоритма происходит выделение памяти под следующие типы данных:

- 2 строки типа str;
- 2 длины строк типа int;
- матрицу с размерами, равными длинам строк, увеличенным на 1;
- адрес возврата - m байт;
- 3 или 4 локальных переменных типа int.

Таким образом, используемая память нерекурсивного алгоритма в среднем равна:

$$M_m = (5,5 + (|s1| + 1) \cdot (|s2| + 1)) \cdot sizeof(int) + 2 \cdot sizeof(str) + m \tag{4.2}$$

Для рекурсивного алгоритма с кешем при каждом вызове происходит выделение памяти под следующие типы данных:

- 2 строки типа str;
- 2 длины строк типа int;
- ссылка на матрицу $*int$;
- адрес возврата - m байт;
- 3 или 4 локальных переменных типа int.

$$M_1 = (5,5 \cdot sizeof(int) + 2 \cdot sizeof(str) + m + sizeof(*int)) \cdot (|s1| + |s2|) \tag{4.3}$$

Также перед началом рекурсии выделяется память под матрицу с размерами, равными длинам строк, увеличенным на 1.

$$M_2 = (|s1| + 1) \cdot (|s2| + 1) \cdot sizeof(int) \tag{4.4}$$

При этом глубина рекурсии равна сумме длин двух строк. Таким образом, используемая память рекурсивного алгоритма с кешем в среднем равна:

$$M_{rc} = M_1 + M_2 \tag{4.5}$$

Алгоритм нахождения расстояния Дамерау-Левенштейна отличается от алгоритма нахождения расстояния Левенштейна только тем, что добавляется операция транспозиции. Поэтому количество локальных переменных для алгоритма Левенштейна будет равно 3. Всё остальное аналогично.

4.5 Вывод

Был проанализирован объем используемой памятия для рекурсивной, матричной и рекурсивной с кешем версий алгоритмов. Можно сделать вывод, что матричные алгоритмы проигрывают рекурсивным, потому что максимальный размер памяти в них растет, как произведение длин строк, а в рекурсивных — как сумма длин строк. При этом рекурсивные алгоритмы с кешем проигрывают обоим.

Приведенные характеристики времени показывают, что рекурсивная реализация алгоритма сильно проигрывает по времени. В связи с этим, рекурсивные алгоритмы следует использовать лишь для малых размерностей строк. Но рекурсивная реализация с кешем намного быстрее классической рекурсивной реализации, поэтому является более оптимальным вариантом для применения.

Также в результате эксперимента было установлено, что алгоритм Дамерау-Левенштейна работает медленнее Левенштейна из-за дополнительной операции.

Заключение

Алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна являются самыми популярными алгоритмами, которые помогают найти редакторское расстояние. В результате выполнения данной лабораторной работы были изучены алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна, построены схемы, соответствующие данным алгоритмам. Реализовано ПО, которое вычисляет редакторское расстояние с помощью разных версий данных алгоритмов. В рамках выполнения лабораторной работы цель достигнута и решены следующие задачи:

- изучены расстояния Левенштейна и Дамерау-Левенштейна;
- реализованы алгоритмы поиска изученных расстояний;
- проведено сравнение временных характеристик, а также затрат по памяти;
- подготовлен отчет о выполненной лабораторной работе.

Список используемых источников

[1] Методика идентификации пассажира по установочным данным [Электронный ресурс]. Режим доступа: https://cyberleninka.ru/article/n/metodika-identifikatsii-passazhira-po-ustanovochnym-dannym/viewer (дата обращения: 10.10.2022).

[2] Welcome to Python [Электронный ресурс]. Режим доступа: https://www.python.org (дата обращения: 10.10.2022).

[3] time — Time access and conversions [Электронный ресурс]. Режим доступа: https://docs.python.org/3/library/time.html#functions (дата обращения: 10.10.2022).

[4] Manjaro Linux [Электронный ресурс]. Режим доступа: https://manjaro.org (дата обращения: 10.10.2022).

[5] Процессор Intel® Core™ i5-10210U [Электронный ресурс]. Режим доступа: https://ark.intel.com/content/www/ru/ru/ark/products/195436/intel-core-i510210u-processor-6m-cache-up-to-4-20-ghz.html (дата обращения: 10.10.2022).