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 ИУ «Информатика и системы управления»

КАФЕДРА ИУ-7 «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ

по лабораторной работе № 1

по курсу «Анализ алгоритмов»

на тему: «Расстояние Левенштейна»

Студент ИУ7-55Б _______________ И. Д. Половинкин
(Подпись, дата)

Преподаватель _______________ Л. Л. Волкова
(Подпись, дата)

2024 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

При наборе текста выявляются трудности из-за опечаток. Возникает необходимость в эффективных средствах для их быстрого исправления.

Для решения подобных проблем в области прикладной лингвистики существует направление, известное как компьютерная лингвистика. В ней разрабатываются и применяются компьютерные программы, предназначенные для исследования языка и моделирования его функционирования в различных условиях [1].

Одним из первых , был советский ученый В. И. Левенштейн [2]. Его алгоритм стал известен как расстояние Левенштейна — метрика, измеряющая различие между двумя строками в количестве редакторских операций (вставки, удаления, замены), необходимых для преобразования одной последовательности символов в другую. Расстояние Дамерау-Левенштейна является модификацией этого алгоритма, добавляя к редакторским операциям еще и транспозицию, обмен двух соседних символов. Эти алгоритмы нашли применение не только в компьютерной лингвистике, но также в биоинформатике для оценки схожести различных участков ДНК и РНК.

Целью данной лабораторной работы является изучение, реализация и исследование алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна. Для достижения поставленной цели необходимо решить следующий набор задач:

– проанализировать алгоритмы Левенштейна и Дамерау-Левештейна;
– формально описать алгоритмы Левенштейна и Дамерау-Левештейна;
– выполнить тестирование реализации алгоритмов методом черного ящика;
– определить зависимость времени выполнения и необходимой памяти для функционирования предлагаемой реализации от размерности входных данных;
– привести рекомендации по использованию алгоритмов.

1 Аналитический раздел

1.1 Расстояние Левенштейна

Расстояние Левенштейна (редакторское расстояние) между двумя строками представляет собой минимальное количество операций вставки, удаления и замены, необходимых для превращения одной строки в другую [2].

При преобразовании одной строки в другую используются следующие операции [2]:

– I (англ. *insert*) – вставка;

– D (англ. *delete*) – удаление;

– R (англ. *replace*) – замена;

– M (англ. *match*) – совпадение.

Штрафом называется стоимость каждой из этих операций. Для вставки, удаления, замены значение принимается 1, для совпадения – 0 [2].

Необходимо найти последовательность замен с минимальным суммарным штрафом.

1.1.1 Рекуррентный алгоритм

Пусть s_1 и s_2 – две строки длиной M и N соответственно над некоторым алфавитом. Расстояние между s_1 и s_2 рассчитывается по рекуррентной формуле (1.1).

$$D(i, j) = \begin{cases} 0, & \text{i = 0, j = 0,} \\ i, & \text{j = 0, i > 0,} \\ j, & \text{i = 0, j > 0,} \\ \min\{D(i, j-1) + 1, & \\ D(i-1, j) + 1, & \text{i > 0, j > 0,} \\ D(i-1, j-1) + f(s_1[i], s_2[j]), & \\ \}. & \end{cases} \tag{1.1}$$

где i, j – длины подстроки $s_1[1..i]$ и $s_2[1..j]$ соответственно, а функция $f(s_1, s_2)$ определяется по формуле (1.2):

$$f(s_1, s_2) = \begin{cases} 0, s_1 = s_2 \\ 1, \text{иначе} \end{cases} . \tag{1.2}$$

1.1.2 Матрица расстояний

При обработке строк с бо́льшими значениями длин M и N реализация алгоритма по рекуррентной формуле (1.1) становится менее эффективной по временным затратам, поскольку требуется многократное вычисление промежуточных результатов. Для оптимизации нахождения расстояния Левенштейна необходимо использовать матрицу стоимостей для хранения этих промежуточных значений. В таком случае алгоритм представляет собой построчное заполнение матрицы значениями $D(i, j)$.

1.1.3 Использование двух строк

Модификацией использования матрицы расстояний является использование только двух строк этой матрицы, в которых хранятся промежуточные значения. После завершения вычислений происходит обмен значениями между этими двумя строками. Далее в ходе выполнения алгоритма перезаписываются значения только второй строки.

1.1.4 Рекурсивный алгоритм с кэшем в форме матрицы

При помощи матрицы можно выполнить оптимизацию рекурсивного алгоритма заполнения. Основная идея такого подхода заключается в том, что при каждом рекурсивном вызове алгоритма выполняется заполнение матрицы стоимостей. Главное отличие данного метода от того, что был описан в разделе 1.1.2 – начальная инициализация матрицы значением ∞. Если рекурсивный алгоритм выполняет вычисления для данных, которые не были обработаны, значение результата минимального расстояния для данного вызова заносится в матрицу. Если рекурсивный вызов уже обрабатывался (ячейка матрицы была заполнена), то алгоритм не выполняет вычислений, а сразу переходит к следующему шагу.

1.2 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна является модификацией расстояния Левенштейна, которая задействует еще одну редакторскую операцию – транспозицию T (англ. *transposition*), которая выполняет обмен соседних символов в слове.

Дамерау показал, что 80% человеческих ошибок при наборе текстов является перестановка соседних символов, пропуск символа, добавление нового символа или ошибочный символ [3]. Таким образом, расстояние Дамерау-Левенштейна часто используется в редакторских программах для проверки правописания. Это расстояние может быть вычислено по рекуррентной формуле (1.3).

$$D(i,j) = \begin{cases} 0, & \text{если i = 0 и j = 0,} \\ i, & \text{если j = 0 и i > 0,} \\ j, & \text{если i = 0 и j > 0,} \\ \min\{D(i,j-1)+1, & \\ D(i-1,j)+1, & \text{если i > 0 и j > 0,} \\ D(i-1,j-1)+f(s_1[i],s_2[j]), & \text{если } s_1[i] = s_2[j-1], \\ D(i-2,j-2)+1, & \text{если } s_1[i-1] = s_2[j], \\ \}, & \\ \min\{D(i,j-1)+1, & \\ D(i-1,j)+1, & \text{иначе,} \\ D(i-1,j-1)+f(s_1[i],s_2[j]), & \\ \}. & \end{cases} \qquad (1.3)$$

где i, j – длины подстроки $s_1[1..i]$ и $s_2[1..j]$ соответственно.

Вывод

В данном разделе были проанализированы алгоритмы Левенштейна и Дамерау-Левенштейна, приведены рекуррентные формулы их вычисления, описаны использование итерационных и рекурсивных случаев выполнения каждого из них.

2 Конструкторский раздел

В данном разделе приводятся схемы итеративного и рекурсивного алгоритмов расстояния Левенштейна, приводится структура программы.

2.1 Алгоритмы Левенштейна

2.1.1 Итеративный алгоритм Левенштейна

На рисунке 2.1 представлен итеративный алгоритм Левенштейна с двумя строками.

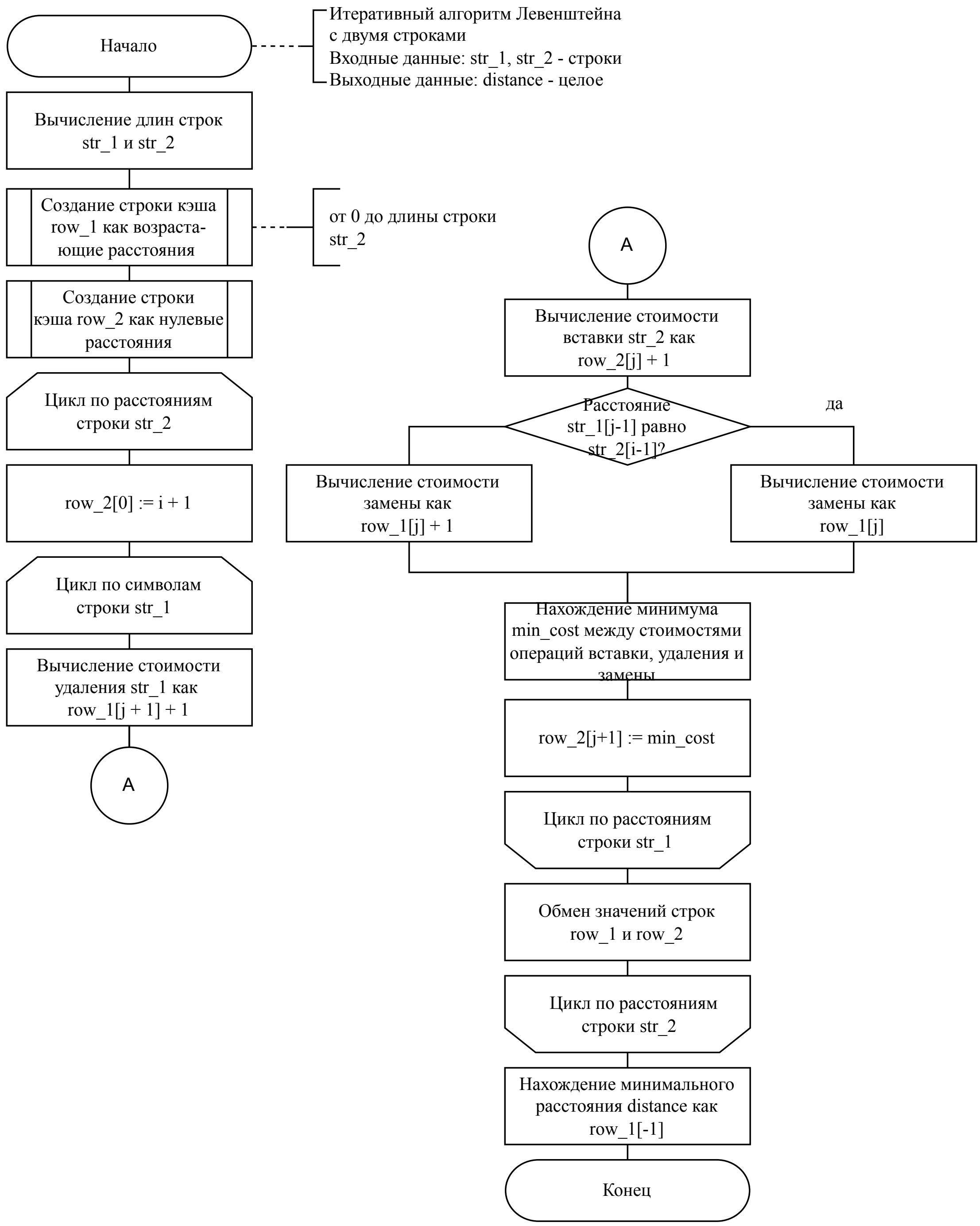

Рисунок 2.1 – Итеративный алгоритм Левенштейна с двумя строками

2.1.2 Рекурсивный алгоритм Левенштейна без кэша

На рисунке 2.2 представлен рекурсивный алгоритм Левенштейна без кэша.

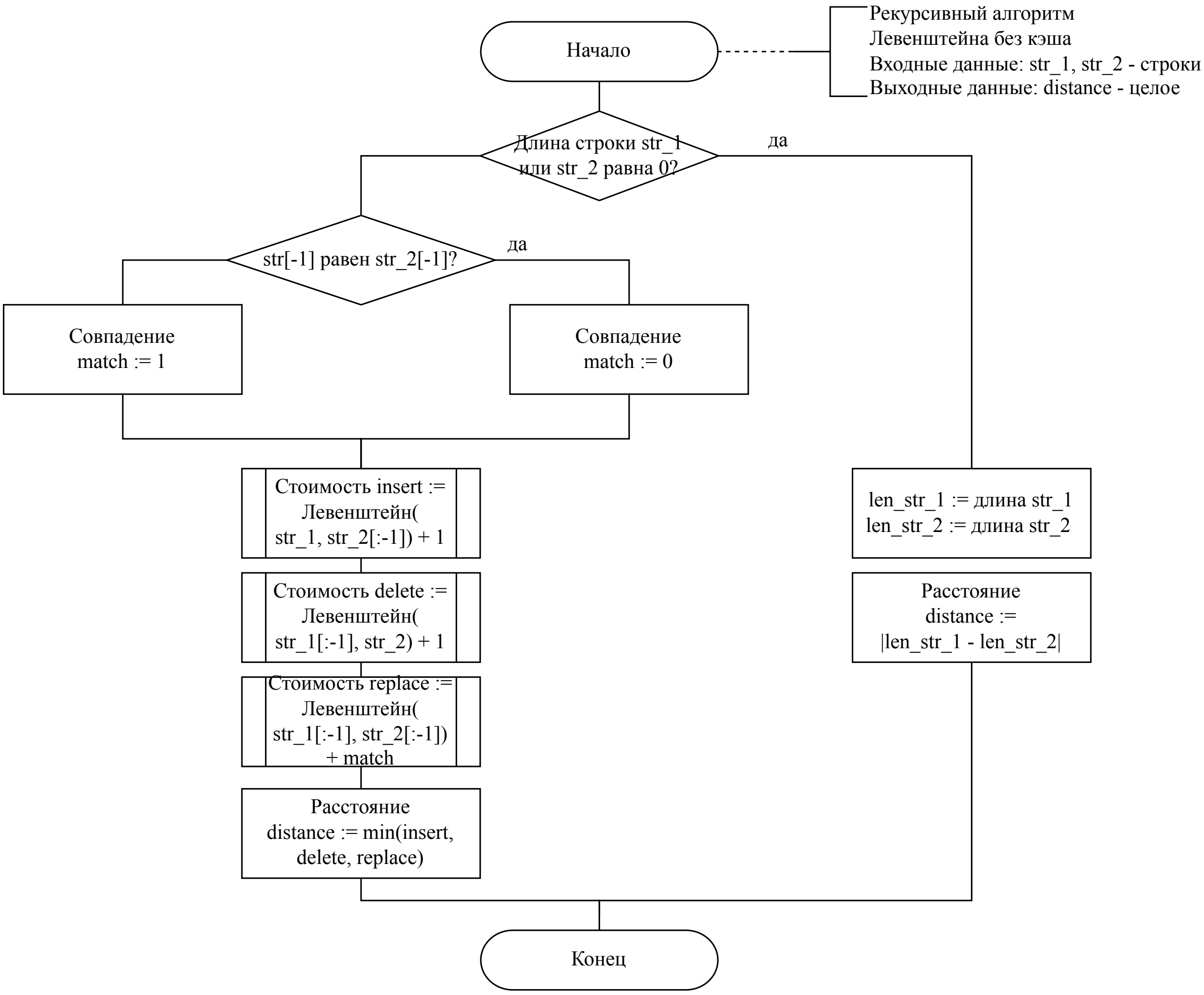

Рисунок 2.2 – Рекурсивный алгоритм Левенштейна без кэша

2.1.3 Рекурсивный алгоритм Левенштейна с матрицей

На рисунке 2.3 и 2.4 представлен рекурсивный алгоритм Левенштейна с матрицей.

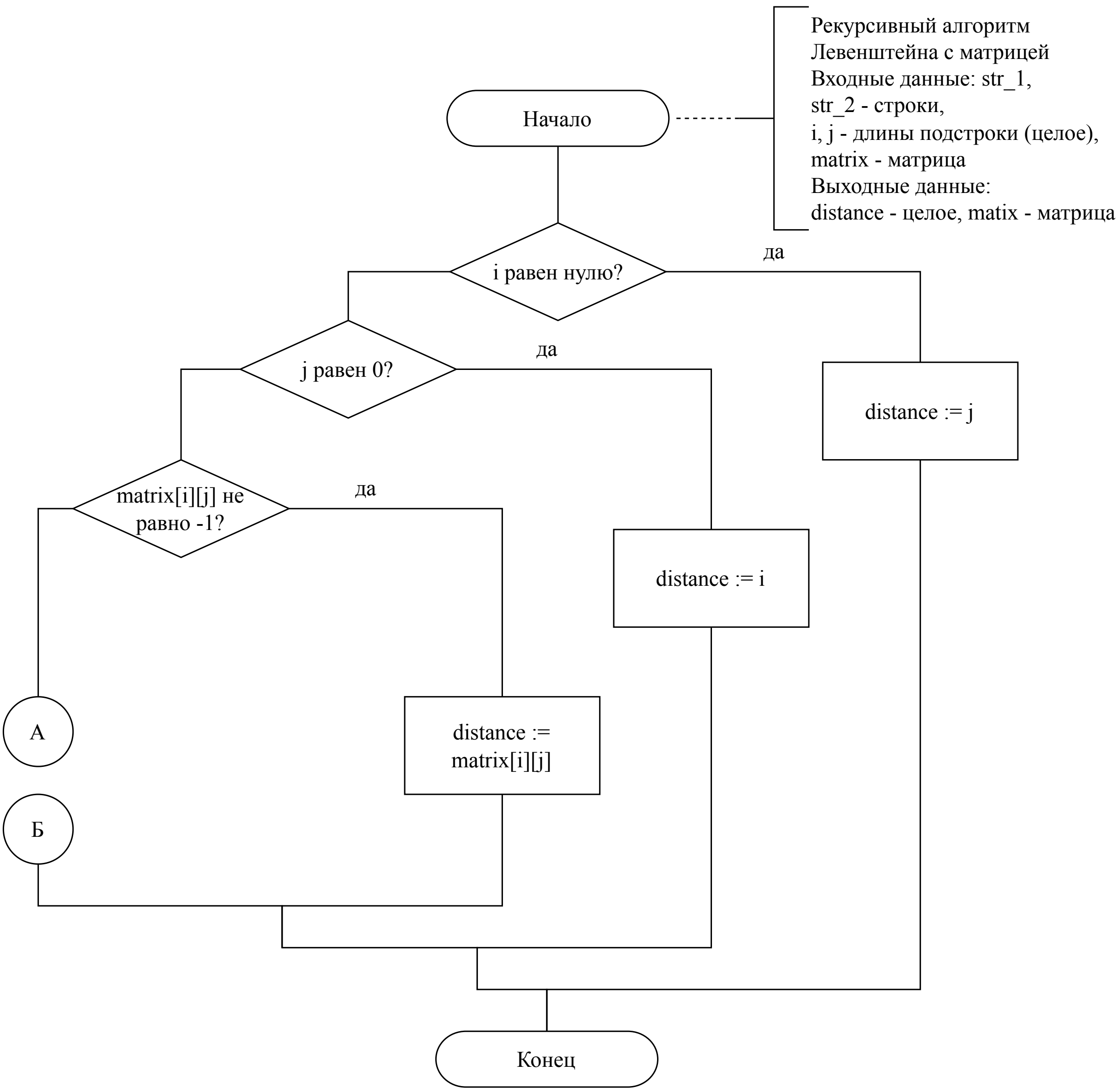

Рисунок 2.3 – Рекурсивный алгоритм Левенштейна с матрицей

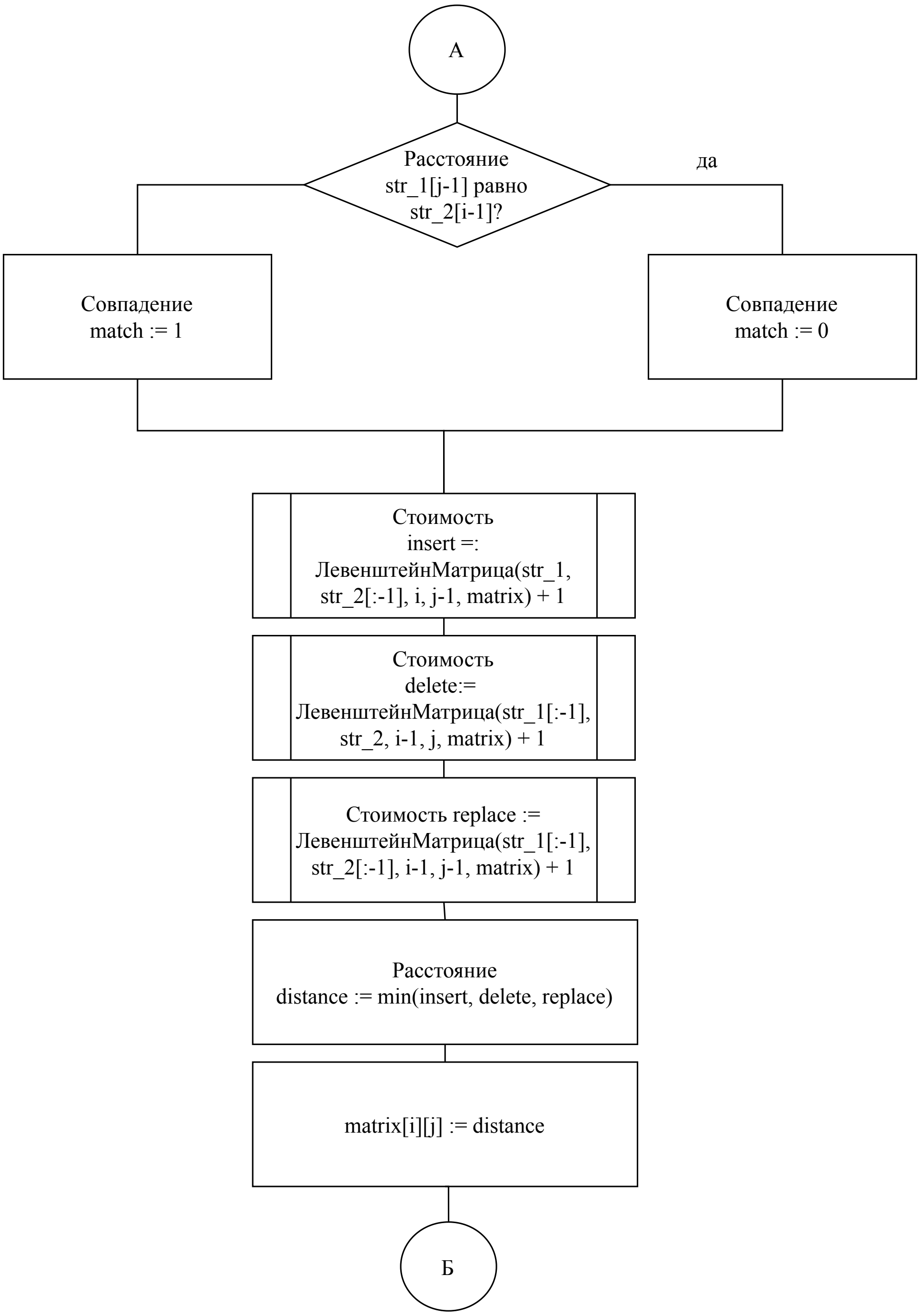

Рисунок 2.4 – Рекурсивный алгоритм Левенштейна с матрицей (продолжение)

2.1.4 Рекурсивный алгоритм Дамерау-Левенштейна

На рисунке 2.5 представлен рекурсивный алгоритм Дамерау-Левенштейна.

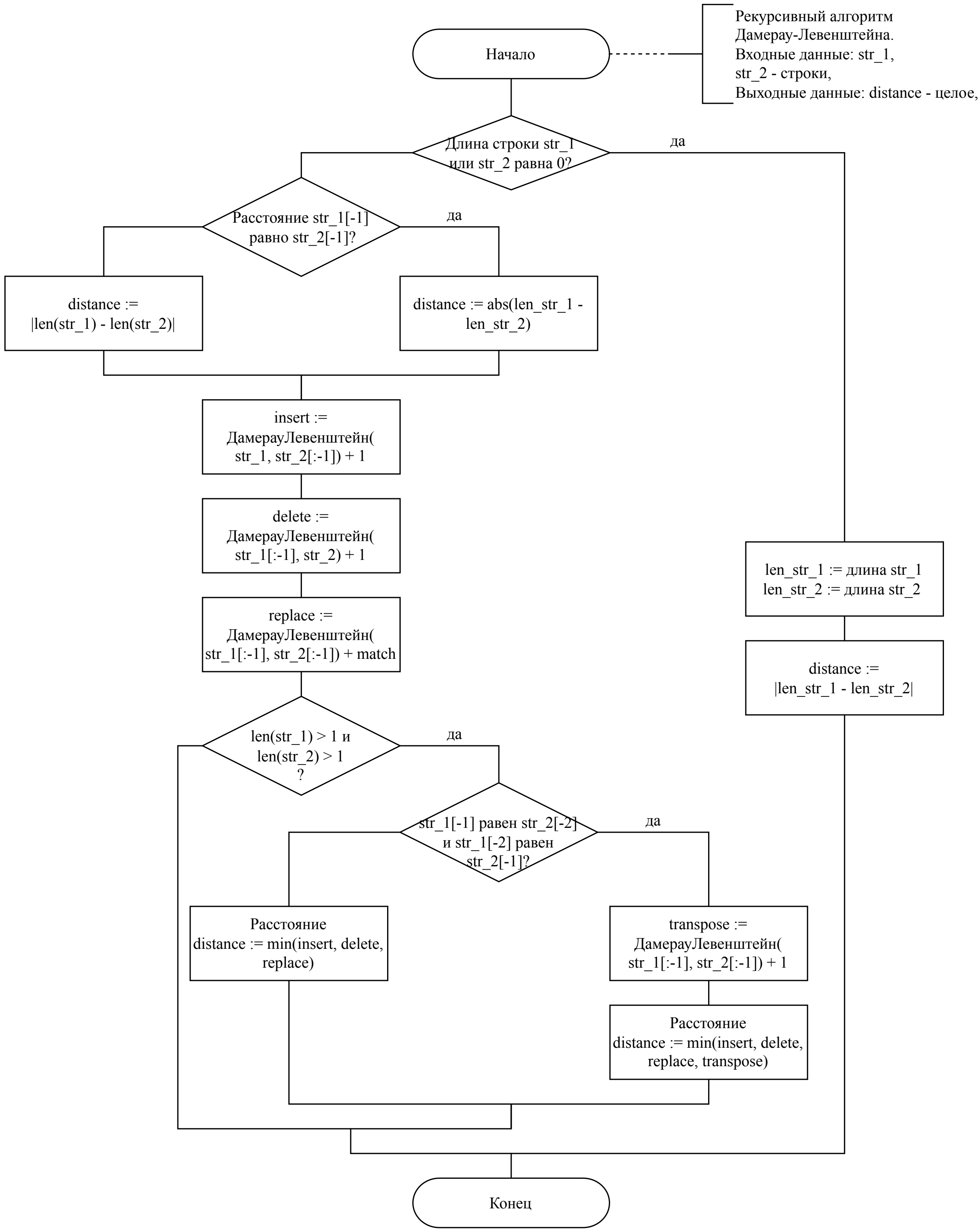

Рисунок 2.5 – Рекурсивный алгоритм Дамерау-Левенштейна

2.2 Структура программы

Программа состоит из следующих модулей:

– main.py: основной файл программы, в котором вызываются алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна;

– distance.py: файл, содержащий код всех представленных алгоритмов;

– test_time.py: замер времени выполнения каждого из алгоритмов;

– generate_string.py: генерация строки заданного размера;

– graph.py: файл отображение результатов замеров зависимости времени работы алгоритмов от входной строки;

– menu.py: перечень команд для взаимодействия с программой.

Вывод

В данном разделе были приведены схемы итеративного и рекурсивного алгоритма Левенштейна и рекурсивного Дамерау-Левенштейна, а также расписана структура программы.

3 Технологический раздел

В данном разделе рассматриваются средства реализации, а также приводятся листинги алгоритмов определения расстояния Левенштейна и Дамерау-Левенштейна

3.1 Средства реализации

В работе для реализации алгоритмов был выбран язык программирования Python [4]. В нем присутствуют библиотека time [5] для замера процессорного времени process_time(), а также для замера используемой памяти при помощи cProfiler() [6].

3.2 Формат входных и выходных данных

Входными данными являются две строки типа str, которые запрашиваются у пользователя. На выходе в результате обработки будет получено число типа int – расстояние Левенштейна. В матричных реализациях алгоритмов Левенштейна используется матрица, являющая двумерным списком типа int.

3.3 Модули программы

3.3.1 Расстояние Левенштейна

Определение расстояния Левенштейна итеративно с использованием двух строк приведено на листинге 3.1.

Листинг 3.1 – Определение расстояния Левенштейна итеративно с использованием двух строк

```
def iterative_levenstein_two_rows(str_1: str, str_2: str) -> int:
    len_str_1 = len(str_1); len_str_2 = len(str_2)
    flag_first_row = 1; flag_second_row = 2

    row_1 = create_row(len_str_1 + 1, flag_first_row)
    row_2 = create_row(len_str_1 + 1, flag_second_row)

    for i in range(len_str_2):
        row_2[0] = i + 1
        for j in range(len_str_1):
            deletion_cost = row_1[j + 1] + 1
            insert_cost = row_2[j] + 1
            replace_cost = row_1[j] if str_1[j - 1] == str_2[i - 1]
                else row_1[j] + 1

            row_2[j + 1] = min(deletion_cost, insert_cost, replace_cost)
        row_1, row_2 = swap_rows(row_1, row_2)
    distance = row_1[-1]
    return distance
```

Определение расстояния Левенштейна рекурсивно без использования кэша приведено на листинге 3.2.

Листинг 3.2 – Определение расстояния Левенштейна рекурсивно без использования кэша

```
def recursive_levenstein(str_1: str, str_2: str) -> int:
    if str_1 == '' or str_2 == '':
        return abs(len(str_1) - len(str_2))
    match = 0 if str_1[-1] == str_2[-1] else 1
    distance = min(recursive_levenstein(str_1, str_2[:-1]) + 1,
                   recursive_levenstein(str_1[:-1], str_2) + 1,
                   recursive_levenstein(str_1[:-1], str_2[:-1]) + match)
    return distance
```

Определение расстояния Левенштейна рекурсивно с использованием матрицы приведено на листинге 3.3.

Листинг 3.3 – Определение расстояния Левенштейна рекурсивно с использованием матрицы

```
def recursive_levenstein_matrix(str_1: str, str_2: str,
    i: int, j: int, matrix: list[list[int]]) \
                            -> Tuple[int, list[list[int]]]:
    if i == 0:
        return j, matrix
    if j == 0:
        return i, matrix
    if matrix[i][j] != -1:
        return matrix[i][j], matrix
    match = 0 if str_1[-1] == str_2[-1] else 1

    insert, matrix = recursive_levenstein_matrix(
                    str_1, str_2[:-1], i, j - 1, matrix)
    delete, matrix = recursive_levenstein_matrix(
                    str_1[:-1], str_2, i - 1, j, matrix)
    replace, matrix = recursive_levenstein_matrix(
                    str_1[:-1], str_2[:-1], i - 1, j - 1, matrix)
    insert += 1; delete += 1; replace += match

    distance = min(insert, delete, replace)
    matrix[i][j] = distance
    return distance, matrix
```

3.3.2 Расстояние Дамерау-Левенштейна

Определение расстояния Дамерау-Левенштейна рекурсивно приведено на листинге 3.4.

Листинг 3.4 – Определение расстояния Дамерау-Левенштейна рекурсивно

```
def recursive_dameray_levenstein(str_1: str, str_2: str) -> int:
    if str_1 == '' or str_2 == '':
        return abs(len(str_1) - len(str_2))
    match = 0 if str_1[-1] == str_2[-1] else 1
    insert = recursive_dameray_levenstein(str_1, str_2[:-1]) + 1
    delete = recursive_dameray_levenstein(str_1[:-1], str_2) + 1
    replace = recursive_dameray_levenstein(str_1[:-1], str_2[:-1]) + match
    if len(str_1) > 1 and len(str_2) > 1 and str_1[-1] == str_2[-2] and \
    str_2[-1] == str_1[-2]:
        distance = min(insert, delete, replace,
            recursive_dameray_levenstein(str_1[:-2], str_2[:-2]) + 1)
    else:
        distance = min(insert, delete, replace)
    return distance
```

3.3.3 Вспомогательные функции

Создание кэша в виде строки приведено на листинге 3.5.

Листинг 3.5 – Создание кэша в виде строки

```
def create_row(len_row: int, flag_row: int) -> list[int]:
    row = list()
    if flag_row == 1:
        for i in range(len_row):
            row.append(i)
    else:
        for i in range(len_row):
            row.append(0)
    return row
```

3.4 Тестирование

Для тестирования используется метод черного ящика. В данном разделе приведена таблица 3.1, в которой указаны классы эквивалентностей тестов.

Таблица 3.1 – Таблица тестов

№	Описание теста	Слово 1	Слово 2	Алгоритм	
				Левенштейн	Дамерау-Левенштейн
1	Пустые строки	”	”	0	0
2	Нет повторяющихся символов	deepcopy	раздел	8	8
3	Инверсия строк	insert	tresni	6	6
4	Два соседних символа	heart	heatr	2	1
5	Одинаковые строки	таблица	таблица	0	0
6	Одна строка меньше другой	город	горо	1	1

Вывод

В данном разделе был обоснован выбор языка программирования, используемых функций библиотек. Реализованы функции, описанные в разделах 1 и 2, проведено их тестирование методом черного ящика по таблице 3.1.

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна. Были выполнено описание каждого из этих алгоритмов, приведены соответствующие математические расчёты.

При тестировании каждого их них и анализе временных характеристик и объема потребляемой памяти можно сделать следующие выводы: выбор алгоритма Дамерау-Левенштейна является оптимальным решением ввиду того, что чаще всего необходимо исправлять ошибки, связанные с обменом двух соседних символов. В ином случае этот алгоритм является проигрышным как по времени, так и по памяти в сравнении с различными реализациями алгоритма Левенштейна. Рекурсивный алгоритм Левенштейна с кэшем в виде матрицы выигрывает по скорости выполнения у данной группы алгоритмов, но он проигрывает по использованию памяти за счет бо́льшего числа вызовов. Таким образом, в ситуациях, не связанных с транспозицей, следует использовать итеративный алгоритм.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. КОМПЬЮТЕРНАЯ ЛИНГВИСТИКА Большая российская энциклопедия - электронная версия [Электронный ресурс]. – Режим доступа, URL: https://bigenc.ru/linguistics/text/2087783 (дата обращения: 05.02.2024)

2. Левенштейн В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов //Доклады Академии наук. – Российская академия наук, 1965. – Т. 163. – №. 4. – С. 845-848.

3. Damerau F. J. A technique for computer detection and correction of spelling errors //Communications of the ACM. – 1964. – Т. 7. – №. 3. – С. 171-176.

4. Python 3.12.1 documentation [Электронный ресурс]. – Режим доступа, URL: https://docs.python.org/3/ (дата обращения: 05.02.2024)

5. time — Time access and conversions — Python 3.12.1 documentation [Электронный ресурс]. – Режим доступа, URL: https://docs.python.org/3/library/time.html (дата обращения: 05.02.2024)

6. The Python Profilers — Python 3.12.1 documentation [Электронный ресурс]. – Режим доступа, URL: https://docs.python.org/3/library/profile.html (дата обращения: 05.02.2024)