Оглавление

Введение

При наборе текста часто возникает проблема, связанная с опечатками. Необходимы средства, которые позволили бы быстро исправлять эти ошибки.

Для решения подобных задач в прикладной лингвистике выделяется такое направление, как компьютерная лингвистика, в которой разрабатываются и используются компьютерные программы, необходимые для исследования языка и моделирования функционирования языка в тех или иных условиях[1].

Одним из первых, кто занялся такой задачей, был советский ученый В.И.Левенштейн [2]. Алгоритм полученного решения связали с его именем. Расстояние Левенштейна - метрика, измеряющая разность двух строк, определяемая в количестве редакторских операций (вставка, удаление, замена), требуемых для преобразования одной последовательности символов в другую. Модификацией данного алгоритма является расстояние Дамерау-Левенштейна, которая добавляет транспозицию, обмен двух соседних символов, к редакторским операциям. Разработанные алгоритмы нашли применение не только в компьютерной лингвистике, но и в биоинформатике для определения схожести разных участков ДНК и РНК.

Целью лабораторной работы является изучение и реализация алгоритмов нахождения расстояний Левенштейна и Дамерлау-Левештейна, а также получения навыка динамического программирования. Для её достижения необходимо выполнить следующие задачи:

- Изучение алгоритмов Левенштейна и Дамерлау-Левештейна
- Разработать данные алгоритмы
- Применение методов динамического программирования для реализации алгоритмов
- Выполнить тестирование реализации алгоритмов методом черного ящика

- Провести сравнительный анализ этих алгоритмов по затратам памяти и процессорному выполнению времени на основе экспериментальных данных

1 Аналитический раздел

1.1 Расстояние Левенштейна

Расстояние Левенштейна (редакторское расстояние) между двумя строками - минимальное количество операций вставки, удаления и замены, необходимых для превращения одной строки в другую.
При преобразовании одной строки в другую, используются следующие операции:

a) I (insert) - вставка;

b) D (delete) - удаление;

c) R (replace) - замена;

Будем считать, что стоимость каждой из этих операции (штраф) равна 1.
Введем еще одну операцию M (match) - совпадение. Её стоимость будет равна нулю. Необходимо найти последовательность замен с минимальным суммарным штрафом.

1.2 Реккурентный алгоритм

Расстояние между двумя строками s1 и s2 рассчитывается по реккуретной формуле 1.2:

$$D(s1[1..i], s2[1..j]) = \begin{cases} 0, & \text{i=0, j=0} \\ j, & \text{i=0, j>0} \\ i, & \text{i>0, j=0} \\ \min \begin{cases} D(s1[1..i], s2[1..j-1]) + 1 \\ D(s1[1..i-1], s2[1..j]) + 1 \\ D(s1[1..i-1], s2[1..j-1]) + \\ f(s1, s2) \end{cases} \end{cases} \tag{1.2.1}$$

Функция f(s1, s2) определяется следующим образом:

$$f(s1, s2) = \begin{cases} 0, \text{ s1=s2} \\ 1, \text{ иначе} \end{cases} \tag{1.2.2}$$

1.3 Матрица расстояний

Реализация формулы (1.2.1) при больших значениях i,j, оказывается менее эффективной по времени ввиду того, что приходится вычислять промежуточные результаты неоднократно. Для оптимизации нахождения расстояния Левенштейна необходимо использовать матрицу стоимостей для хранения этих промежуточных значений. Таким образом, будет необходимо выполнять только построчное заполнение такой матрицы.

1.4 Использование двух строк

Модификацией использованию матрицы является использование только двух строк этой матрицы, в которых хранятся промежуточные зна-

чения. После выполнения вычислений выполняется обмен значений этих двух строк. Алгоритм продолжает работать и перезаписывать значения только второй строки.

1.5 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 (1.3) - 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.

1.6 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна является модификацией расстояния Левенштейна, которая задействует еще одну редакторскую операцию - транспозицию T (transposition. Она 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 следующей реккуретной формуле:

$$D(s1[1..i], s2[1..j]) = \begin{cases} 0, & \text{i=0, j=0} \\ j, & \text{i=0, j>0} \\ i, & \text{i>0, j=0} \\ \min \begin{cases} D(s1[1..i], s2[1..j-1]) + 1 \\ D(s1[1..i-1], s2[1..j]) + 1 \\ D(s1[1..i-1], s2[1..j-1]) + f(s1, s2) \\ D(s1[1..i-1], s2[1..j-1]) + 1, \\ \text{i,j} > 1, a_i = b_{j-1},\ a_{i-1} = b_j \\ \infty, \text{ иначе} \end{cases} \end{cases} \tag{1.5.1}$$

Вывод

В данном разделе были рассмотрены основные способы нахождения редакторского расстояния между двумя строками. Формулы для нахождения расстояния Левенштейна и Дамерау-Левенштейна задаются рекуретно, следовательно, алгоритмы могут быть реализованы как рекурсивно, так и итерационно.

2 Конструкторская часть

2.1 Схемы алгоритмов Левенштейна

Ниже представлены следующие схемы алгоритмов:

рис 2.1 - схема алгоритма итеративного алгоритма Левенштейна с использованием двух строк

рис 2.2 - схема алгоритма рекурсивного алгоритма Левенштейна без кэша

рис 2.3 - схема алгоритма рекурсивного алгоритма Дамерау-Левенштейна с использованием матрицы

рис 2.4 - схема алгоритма итеративного алгоритма Левенштейна с использованием двух строк

Вывод

На основе теоретических данных, полученных в аналатическом разделе, были построены схемы нужных алгоритмов.

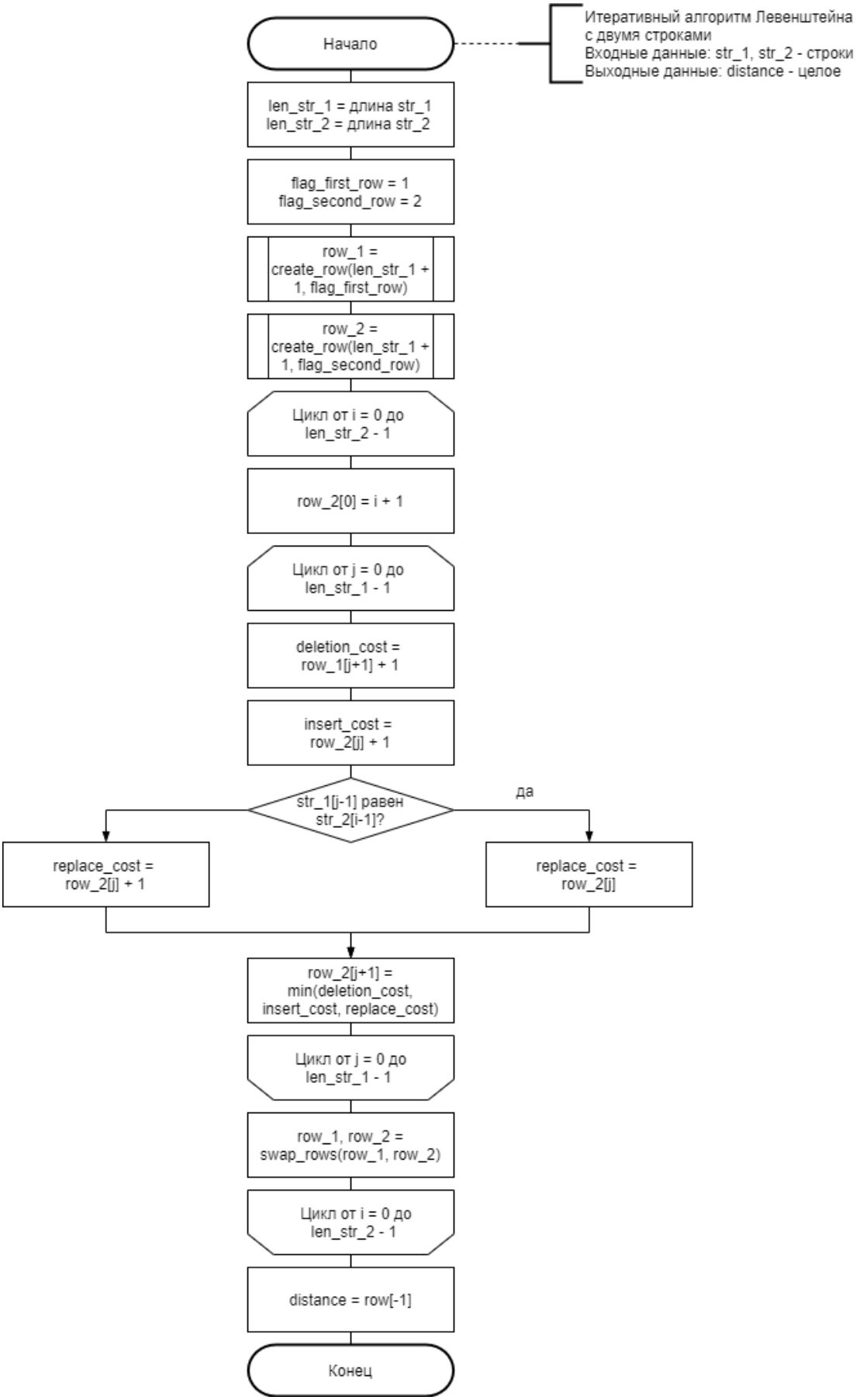

Рисунок 2.1 – Сравнение времени работы алгоритмов Левенштейна.

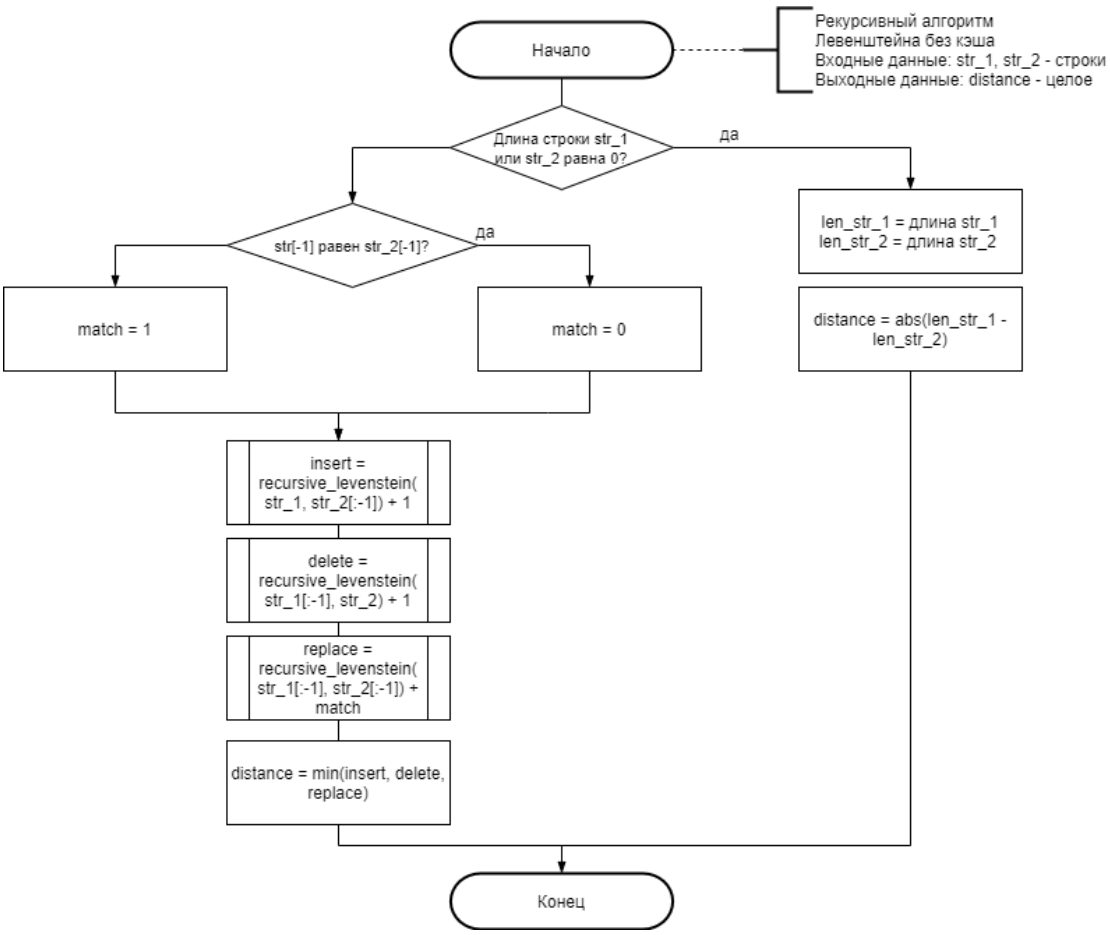

Рисунок 2.2 – Сравнение времени работы алгоритмов Левенштейна.

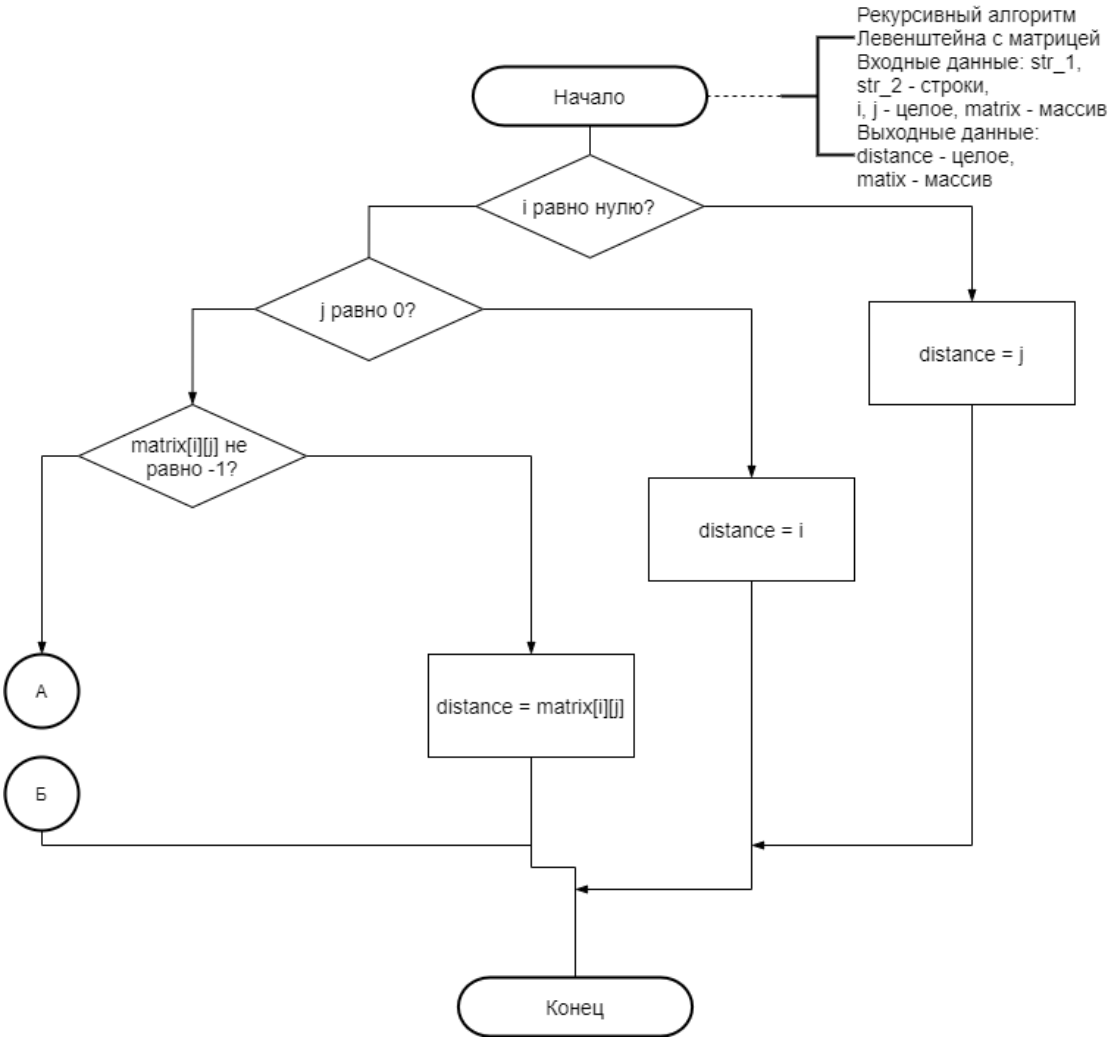

Рисунок 2.3 – Сравнение времени работы алгоритмов Левенштейна.

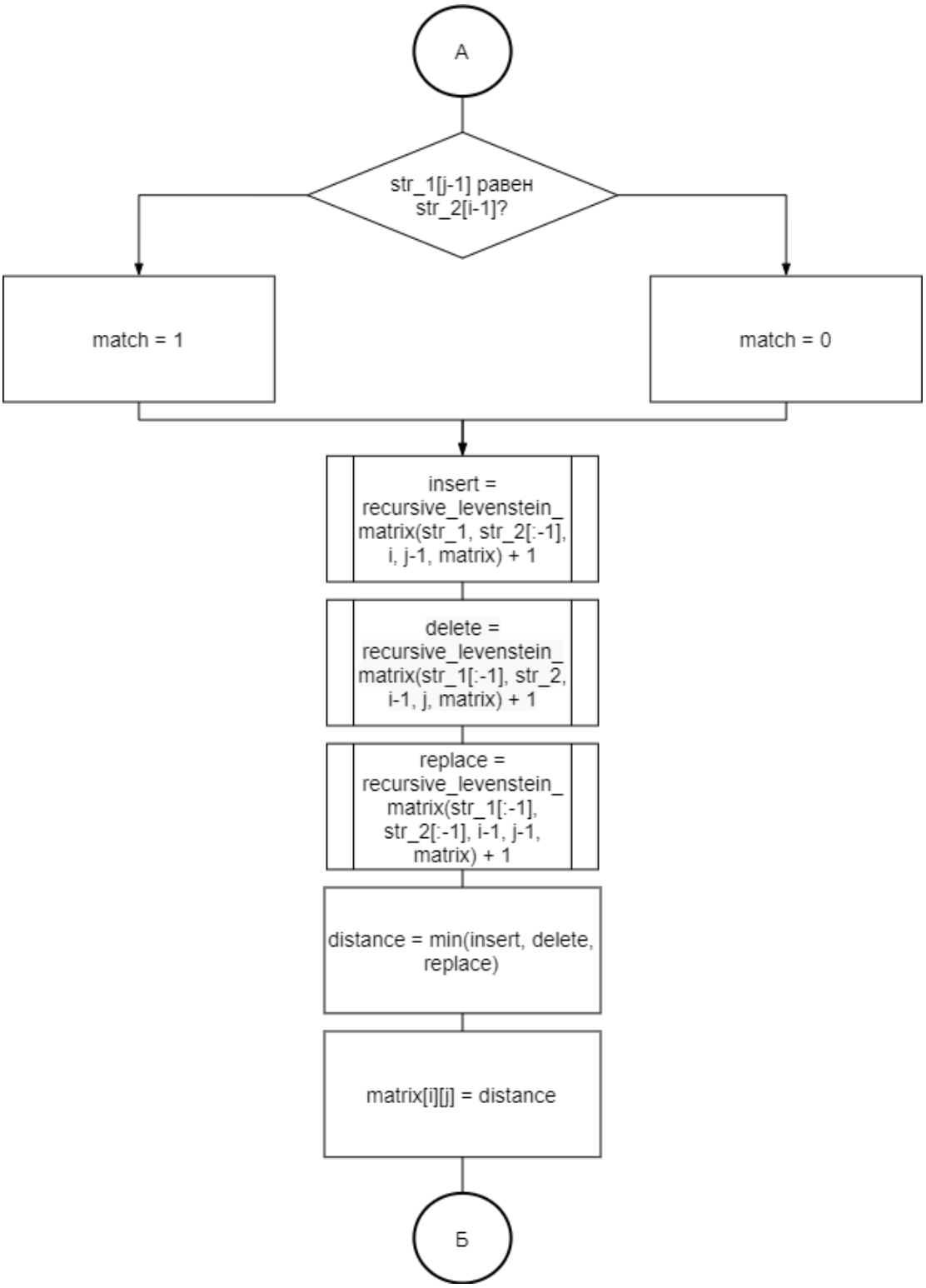

Рисунок 2.4 – Сравнение времени работы алгоритмов Левенштейна.

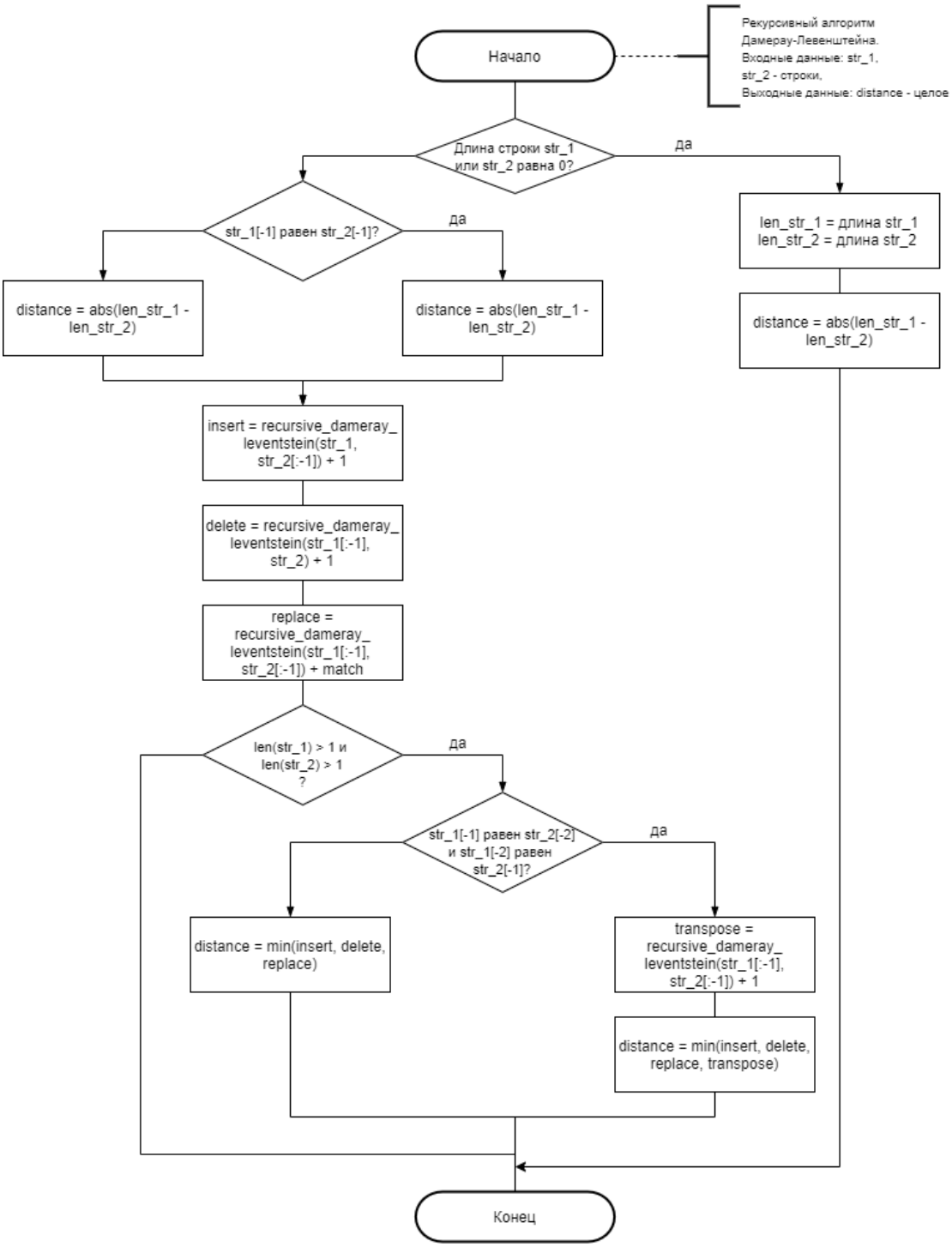

Рисунок 2.5 – Сравнение времени работы алгоритмов Левенштейна.

3 Технологическая часть

3.1 Требования к программному обеспечению

Программа должна отвечать следующим требованиям:

- На вход программе подаются две строки на русском или английском языке в любом регистре;
- Осуществляется выбор алгоритма нахождения расстояния из меню;
- На выходе программа выдает результат - найденное расстояние между двумя строками выбранным пользователем алгоритмом;

3.2 Выбор средств реализации

Для реализации алгоритмов в данной лабораторной работе был выбран язык программирования Python[4]. Он является кроссплатформенным. Имеется опыт разработки на этом языке. В качестве среды разработки был использован Visual Studio Code[5], так как в нем можно работать как на операционной системе Windows, так и на дистрибутивах Linux. При замере процессорного времени был использован модуль test_time[5].

3.3 Листинги программ

Ниже представлены листинги разработанных алгоритмов Левенштейна и Дамерау-Левенштейна.

Листинг 3.1 – Программный код нахождения расстояния Левенштейна итеративно с использованием двух строк

```
1 def iterative_levenstein_two_rows(str_1: str, str_2: str) -> int:
2     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
```

Листинг 3.2 – Программный код нахождения 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
```

Листинг 3.3 – Программный код нахождения расстояния Левенштейна рекурсивно с использованием матрицы

```
    def recursive_levenstein_matrix(str_1: str, str_2: str, i: int, j: int,
        matrix: list) -> 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],
```

```
                                                    -1, j-1, matrix)
    insert += 1; delete += 1; replace += match

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

Листинг 3.4 – Программный код нахождения 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
```

3.4 Утилиты

На листингах представлены программные модули, которые используются в данных функциях:

Листинг 3.5 – Программный код создания для 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
```

Листинг 3.6 – Программный код создания для кэша в виде строки

```
def swap_rows(row_1: list[int], row_2: list[int]) \
                -> Tuple[list[int], list[int]]:
    temp_row = list()
    temp_row = deepcopy(row_1)
    row_1 = deepcopy(row_2)
    row_2 = deepcopy(temp_row)
    return row_1, row_2
```

3.5 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 выбран язык программирования, среда разработки. Реализованы функции, описанные в аналитическом разделе, и проведено их тестирование методом черного ящика по таблице 3.1.

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристии устройства, на котором выполнялось тестирование:

- Операционная система: Windows 10 Pro
- Память: 8 GiB
- Процессор: Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz 1.80 GHz

Тестирование проводилось на ноутбуке, который был подключен к сети питания. Во время проведения тестирования ноутбук был нагружен только встроенными приложениями окружения, самим окружением и системой тестирования

4.2 Временные харастеристики выполнения

Замер процессорного времени работы алгоритмов производилось при помощи модуля time функцией process_time().

Проведем анализ времени работы алгоритмов. Исходными данными будут случайно сгенерированные строки длиной 3, 4, 5, 6, 7, 8. Единичные замеры выдадут крайне маленький результат, поэтому проведем работу каждого алгоритма n = 1000 раз и поделим на число n. Получим среднее значение работы каждого из алгоритмов. Результат приведен на рис 4.1:

Как видно из результатов, рекурсивный алгоритм Левенштейна без кэша и алгоритм Дамерау-Левенштейн имеют большой асимпотический рост, начиная уже со строки длиной 7. Последний имеет наибольший рост. Это объясняется тем, что этот алгоритм задействует дополнительную операцию - транспонирование, которая тоже приводит к вызову рекурсии.

Выполнив анализ двух остальных алгоритмов на значения входных строк

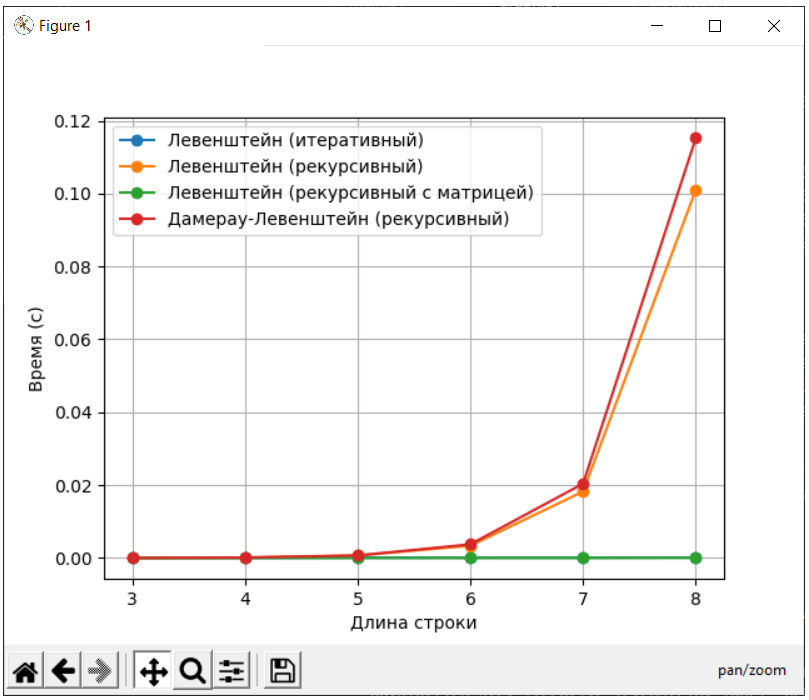

Рисунок 4.1 – Сравнение времени работы алгоритмов Левенштейна.

длиной 25, 50, 75, 100, 125, 150, получим слежующий результат, представленный на рис 4.2:

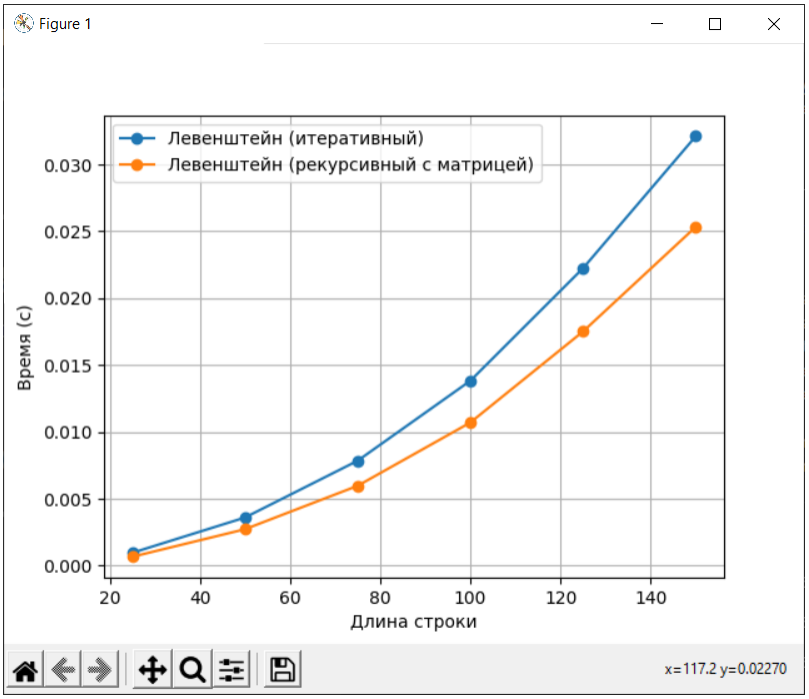

Рисунок 4.2 – Сравнение времени работы рекурсивного и некурсивного алгоритмов Левенштейна.

Рекурсивный алгоритм Левенштейна с использованием матрицы выполняется быстрее, нежели чем итеративный с использованием двух строк. Это объясняется тем, что в итеративном случае выполняется дополни-

тельная операция по обмену значений двух строк. На это необходимо дополнительное время.

4.3 Объем потребляемой памяти

Замеры используемой памяти и число вызовов рекурсии проводились при помощи модуля memory_profiler. При исходных строках, длинною 3, требуется 52,8 Mb памяти. Результаты вызовов и объем потребляемой памяти приведены в таблице (4.3.1):

Таблица 4.1 – Число вызовов каждого алгоритма

Левенштейн			Дамерау-Левенштейн
Итеративный с двумя строками	Рекурсивный без кэша	Рекурсивный с матрицей	Рекурсивный
1	94	28	94

Общее значение потребляемой памяти складывается по формуле 4.3:

$$S = n\_calls * V \tag{4.3.1}$$

где:

- n_calls - число вызовов функций
- V - объем памяти, занимаемый одним вызовом функции

По результатам исследования памяти алгоритм Левенштейна и Дамерау-Левенштейна потребляют больше всего памяти при работе. Итеративный алгоритм Левенштейна с двумя строками занимает меньше всего памяти

Вывод

Характеристики алгоритмов, приведенные в разделах (4.2) и (4.3) позволяют сделать вывод о том, что рекурсивный вызов Левенштейна без кэша и Дамерау-Лвенштейна проигрывают как по скорости, так и

по памяти итеративному. Причем рекурсивный алгоритм Левенштейна с матрицей работает быстрее, чем итеративный с двумя строками, но также проигрывает ему по памяти.

Сравнивая между собой рекурсивные вызовы алгоритмов Левенштейна и Дамерау-Левенштейна, можно сделать вывод о том, что рекурретный алгоритм поиска расстояния Левенштейна с матрицей выигрывает как по времени, так и по памяти, а рекуррентный Дамерау-Левенштейн проигрывает по обоим параметрам. Однако, стоит отметить, что в системах автоматического исправления текста, где чаще всего встречаются ошибки, связанные с транспозицией двух символов, алгоритм Дамерау-Левенштейна будет наиболее оптимальным.

Заключение

В ходе выполнения лабораторной работы были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна. Были выполнено описание каждого из этих алгоритмов, приведены соотвествующие математические расчёты. Были получены навыки динамического программирования, а также реализованны данные алгоритмы. При тестировании каждого их них и анализе временных характеристик и объем потребляемой памяти можно сделать следующие выводы: алгоритм Дамерау-Левенштейна является наиболее оптимальным 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. Самым быстрым по времени является рекурсивный алгоритм Левенштейн с кэшем в виде матрицы; но он достаточно много потребляет памяти за счет большого числа вызовов. Поэтому в иных ситуациях, не связанных с транспозицей, следует использовать итеративный алгоритм.