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»

НАПРАВЛЕНИЕ ПОДГОТОВКИ «09.03.04 Программная инженерия»

ОТЧЕТ

ПО ЛАБОРАТОРНОЙ РАБОТЕ №1

Название: Расстояние Левенштейна и Дамерау – Левенштейна

Дисциплина: Анализ алгоритмов

Студент	ИУ7-53Б		И. О. Артемьев
	Группа	Подпись, дата	И. О. Фамилия
Преподаватель			Л. Л. Волкова
		Подпись, дата	И. О. Фамилия

Москва, 2021 г.

Содержание

Страница

Введение

В настоящее время перед компьютерной лингвистикой ставится множество задач. Одна из них - поиск редакционного расстояния между строками. Это определение минимального количества редакционных операций, необходимых для превращения одной строки в другую. Впервые эту задачу обозначил В. И. Левенштейн, имя которого закрепилось за ней.

При вычислении расстояния Левенштейна редакционные операции ограничиваются вставкой, удалением и заменой. В случае расстояния Дамерау - Левенштейна к операциям добавляется транспозиция - перестановка двух соседних символов.

Данные алгоритмы находят применение не только в компьютерной лингвистике для исправления ошибок или автозамены слов, но также в биоинформатике для определения разных участков ДНК и РНК.

Существует множество модификаций упомянутых алгоритмов. В данной работе будут рассмотрены лишь те, которые используют парадигмы динамического программирования.

Целью данной работы является реализация и изучение следующих алгоритмов нахождения редакционного расстояния:

- рекурсивный Левенштейн;
- кеширующий Левенштейн;
- рекурсивный Дамерау - Левенштейна;
- кеширующий Дамерау - Левенштейна.

Для достижения поставленной цели необходимо выполнить следующие задачи:

- изучить выбранные алгоритмы нахождения редакционного расстояния;
- составить схемы рассмотренных алгоритмов;
- реализовать разработанные алгоритмы нахождения редакционного расстояния;

- провести сравнительный анализ алгоритмов по затрачиваемым ресурсам (время и память);
- описать и обосновать полученные результаты.

1. Аналитический раздел

1.1 Расстояние Левенштейна

Редакторское расстояние (расстояние Левенштейна) – это минимальное количество операций вставки, удаления и замены, необходимых для превращения одной строки в другую. Каждая редакторская операция имеет цену (штраф). В общем случае, имея на входе строку $X = x_1x_2...x_n$ и $Y = y_1y_2...y_n$, расстояние между ними можно вычислить с помощью операций:

- $\text{delete}(u, \varepsilon) = \delta$
- $\text{insert}(\varepsilon, v) = \delta$
- $\text{replace}(u, v) = \alpha(u, v) \leq 0$ (здесь, $\alpha(u, u) = 0$ для всех u).

Необходимо найти последовательность замен с минимальным суммарным штрафом. Далее, цена вставки и удаления будет считаться равной 1.

1.2 Рекурсивная формула

Используя условные обозначения, описанные в разделе 1.1, рекурсивная формула для нахождения расстояния Левенштейна $f(i, j)$ между подстроками $x_1...x_i$ и $y_1...y_j$ имеет следующий вид:

$$f_{X,Y}(i,j) = \begin{cases} \delta_i & j = 0 \\ \delta_j & i = 0 \\ min \begin{cases} \alpha(x_i, y_i) + f_{X,Y}(i-1, j-1) \\ \delta + f_{X,Y}(i-1, j) \\ \delta + f_{X,Y}(i, j-1) \end{cases} & \text{иначе.} \end{cases} \tag{1.1}$$

$f_{X,Y}$ – редакционное расстояние между двумя подстроками – первыми i

символами строки X и первыми j символами строки Y. Очевидны следующие утверждения:

- Если редакционное расстояние нулевое, то строки равны:
  $f_{X,Y} = 0 \Rightarrow X = Y$
- Редакционное расстояние симметрично:
  $f_{X,Y} = f_{Y,X}$
- Максимальное значение $f_{X,Y}$ – размерность более длинной строки:
  $f_{X,Y} \leq max(|X|, |Y|)$
- Минимальное значение $f_{X,Y}$ – разность длин обрабатываемых строк:
  $f_{X,Y} \geq abs(|X| - |Y|)$
- Аналогично свойству треугольника, редакционное расстояние между двумя строками не может быть больше чем редакционные расстояния каждой из этих строк с третьей:
  $f_{X,Y} \leq f_{X,Z} + f_{Z,Y}$

1.3 Матрица расстояний

В 2001 году был предложен подход, использующий динамическое программирование. Этот алгоритм, несмотря на низкую эффективность, один из самых гибких и может быть изменен в соответствии с функцией нахождения расстояния, по которой производится расчет.

Пусть $C_{0..|X|,0..|Y|}$ – матрица расстояний, где $C_{i,j}$ – минимальное количество редакторских операций, необходимое для преобразования подстроки $x_1...x_i$ в подстроку $y_1...y_j$. Матрица заполняется следующим образом:

$$Ci,j = \begin{cases} i & j = 0 \\ j & i = 0 \\ min \begin{cases} C_{i-1,j-1} + \alpha(x_i, y_i), & \\ C_{i-1,j} + 1, & \text{иначе.} \\ C_{i,j-1} + 1) & \end{cases} \end{cases} \tag{1.2}$$

При решении данной задачи используется ключевая идея динамического программирования – чтобы решить поставленную задачу, требуется решить отдельные части задачи (подзадачи), после чего объединить решения подзадач в одно общее решение. Здесь небольшие подзадачи – это заполнение ячеек таблицы с индексами $i < |X|, j < |Y|$. После заполнения всех ячеек матрицы в ячейке $C_{|X|,|Y|}$ будет записано искомое расстояние.

1.4 Расстояние Дамерау – Левенштейна

Расстояние Дамерау – Левенштейна – модификация расстояния Левенштейна, добавляющая транспозицию к редакторским операциям, предложенными Левенштейном (см. 1.1). изначально алгоритм разрабатывался для сравнения текстов, набранных человеком (Дамерау показал, что 80% человеческих ошибок при наборе текстов составляют перестановки соседних символов, пропуск символа, добавление нового символа, и ошибка в символе. Поэтому метрика Дамерау-Левенштейна часто используется в редакторских программах для проверки правописания).

Используя условные обозначения, описанные в разделе 1.1, рекурсивная формула для нахождения расстояния Дамерау – Левенштейна $f(i, j)$ между подстроками $x_1...x_i$ и $y_1...y_j$ имеет следующий вид:

$$f_{X,Y}(i,j) = \begin{cases} \delta_i & j = 0 \\ \delta_j & i = 0 \\ min \begin{cases} \alpha(x_i, y_i) + f_{X,Y}(i-1, j-1) \\ \delta + f_{X,Y}(i-1, j) \\ \delta + f_{X,Y}(i, j-1) \\ \begin{cases} \delta + f_{X,Y}(i-2, j-2) & i,j > 1 x_i = y_{j-1} x_{i-1} = y_j \\ \infty & \text{иначе;} \end{cases} \end{cases} & \text{иначе.} \end{cases} \tag{1.3}$$

1.5 Вывод

В данном разделе были рассмотренны принципы работы алгоритмов нахождения редакционного расстояния. Полученных знаний достаточно для разработки выбранных алгоритмов.

В качестве входных данных в программу будут подаваться две строки, на выходных данных будет редакционное расстояние.

Реализуемое ПО будет работать в пользовательском режиме (вывод редакционного расстояния), а также в экспериментальном (проведение замеров времени выполнения алгоритмов).

2. Конструкторский раздел

В данном разделе будут спроектированы схемы алгоритмов, произведена оценка трудоемкости алгоритмов, описаны используемые типы данных, а также произведена оценка памяти и описана структура ПО.

2.1 Схемы алгоритмов

На рисунках 2.1 - 2.4 представлены схемы рассматриваемых алгоритмов.

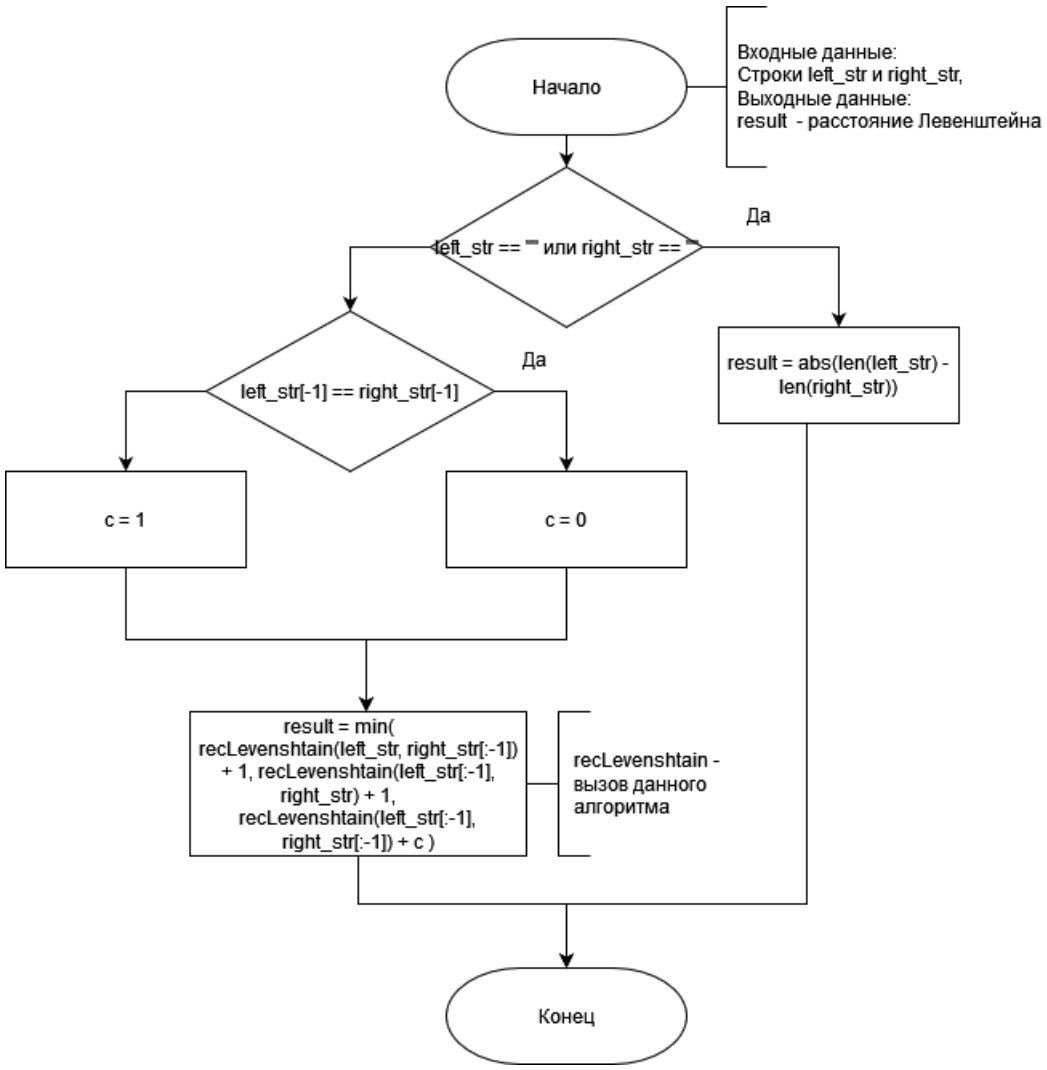

Рисунок 2.1 – Схема рекурсивного алгоритма поиска расстояния Левенштейна

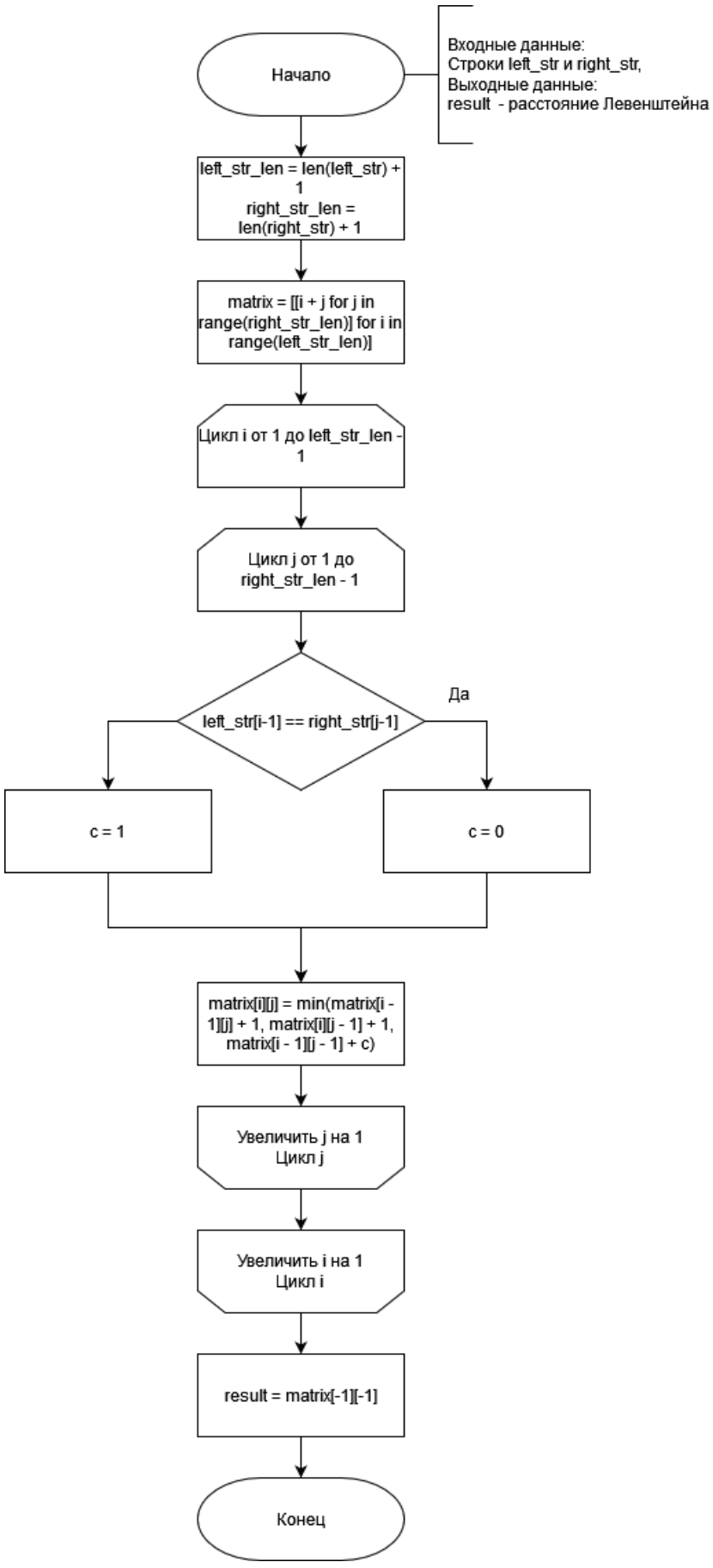

Рисунок 2.2 – Схема кеширующего алгоритма поиска расстояния Левенштейна

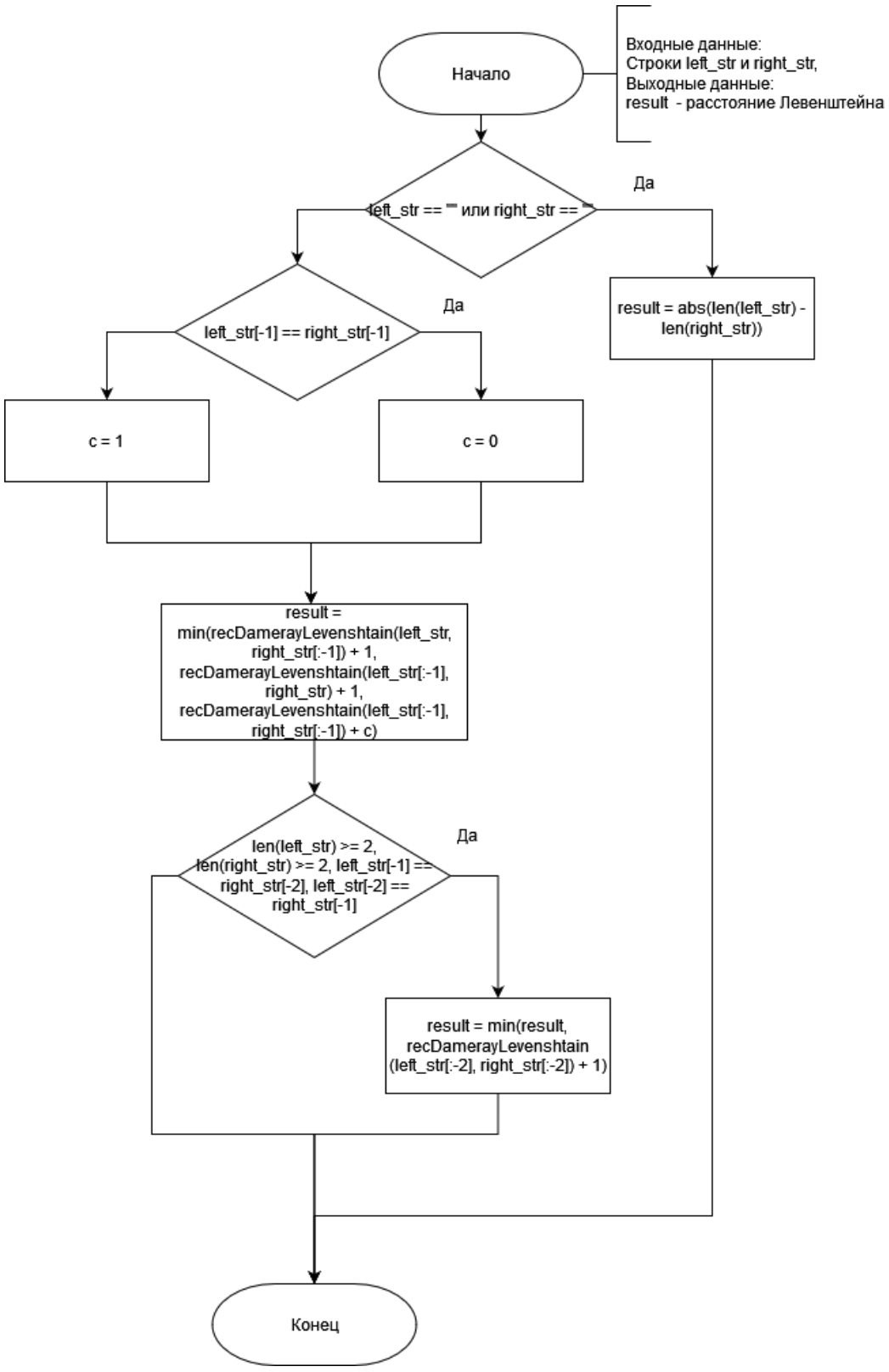

Рисунок 2.3 – Схема рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна

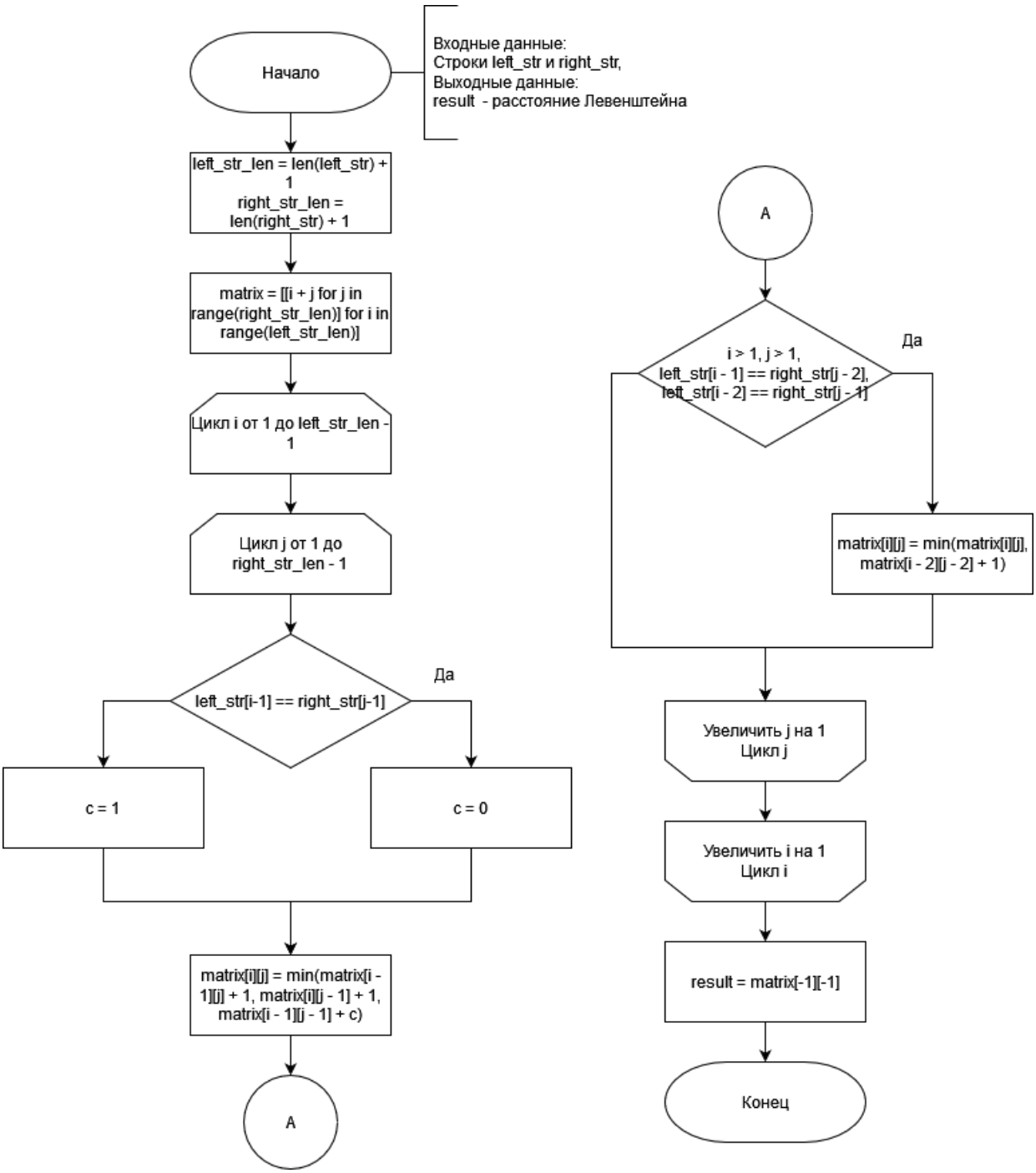

Рисунок 2.4 – Схема кеширующего алгоритма поиска расстояния Дамерау-Левенштейна

2.2 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие структуры данных:

- строка типа str заданного размера;
- длина строки - целое число типа int;
- кэш в форме матрицы - матрица типа int.

2.3 Структура ПО

ПО будет состоять из следующих модулей:

- main.py - модуль, вызывающий загрузку меню (является модулем запуска);
- menu.py - модуль, содержащий меню пользователя;
- myio.py - модуль, содержащий функции формирования строк;
- edit_dist.py - модуль, содержащий функции подсчета редакционного расстояния;
- plot.py - модуль, содержащий функции для построения графиков сложности подсчета редакционного расстояния.

2.4 Вывод

На основе полученных в аналитическом разделе знаний об алгоритмах были спроектированы схемы алгоритмов, выбраны используемые типы данных, а также описана структура ПО.

3. Технологический раздел

В данном разделе будут описаны средства реализации ПО и листинга кода алгоритмов, а также рассмотрены тестовые случаи.

3.1 Средства реализации

Для реализации программ я выбрал язык программирования Python, так как я очень хорошо знаком с этим языком и пишу на нем давно, также этот язык является удобным, безопасным и в нем присутствуют инструменты замера времени.

3.2 Листинги кода

В листингах 3.1 - 3.4 приведены реализации алгоритмов, изученных в аналитическом разделе.

Листинг 3.1 – Рекурсивный Левенштейн

```
def recLevenshtain(left_str: str, right_str: str) -> int:
    if left_str == "" or right_str == "":
        return abs(len(left_str) - len(right_str))

    if left_str[-1] == right_str[-1]:
        c = 0
    else:
        c = 1

    return min(
        recLevenshtain(left_str, right_str[:-1]) + 1,
        recLevenshtain(left_str[:-1], right_str) + 1,
        recLevenshtain(left_str[:-1], right_str[:-1]) + c,
    )
```

Листинг 3.2 – Кеширующий Левенштейн

```
def cacheLevenshtain(left_str: str, right_str: str) -> int:
    left_str_len = len(left_str) + 1
    right_str_len = len(right_str) + 1
    matrix = [[i + j for j in range(right_str_len)] \
```

```
                        for i in range(left_str_len)]

    for i in range(1, left_str_len):
        for j in range(1, right_str_len):
            if left_str[i - 1] == right_str[j - 1]:
                c = 0
            else:
                c = 1

            matrix[i][j] = min(
                matrix[i - 1][j] + 1, matrix[i][j - 1] + 1,\
                matrix[i - 1][j - 1] + c
            )

    return matrix[-1][-1]
```

Листинг 3.3 – Рекурсивный Дамерау-Левенштейн

```
def recDamerayLevenshtain(left_str: str, right_str: str) -> int:
    if left_str == "" or right_str == "":
        return abs(len(left_str) - len(right_str))

    if left_str[-1] == right_str[-1]:
        c = 0
    else:
        c = 1

    result = min(
        recDamerayLevenshtain(left_str, right_str[:-1]) + 1,
        recDamerayLevenshtain(left_str[:-1], right_str) + 1,
        recDamerayLevenshtain(left_str[:-1], right_str[:-1]) + c,
    )

    if (
        len(left_str) >= 2
        and len(right_str) >= 2
        and left_str[-1] == right_str[-2]
        and left_str[-2] == right_str[-1]
    ):
        result = min(result, recDamerayLevenshtain(left_str[:-2], \
                                                   right_str[:-2]) + 1)

    return result
```

Листинг 3.4 – Кеширующий Дамерау-Левенштейн

```
def cacheDamerayLevenshtain(left_str: str, right_str: str) -> int:
    left_str_len = len(left_str) + 1
```

```
    right_str_len = len(right_str) + 1
    matrix = [[i + j for j in range(right_str_len)]\
                     for i in range(left_str_len)]

    for i in range(1, left_str_len):
        for j in range(1, right_str_len):
            if left_str[i - 1] == right_str[j - 1]:
                c = 0
            else:
                c = 1

            matrix[i][j] = min(
                matrix[i - 1][j] + 1, matrix[i][j - 1] + 1,\
                matrix[i - 1][j - 1] + c
            )

            if (
                (i > 1 and j > 1)
                and left_str[i - 1] == right_str[j - 2]
                and left_str[i - 2] == right_str[j - 1]
            ):
                matrix[i][j] = min(matrix[i][j], matrix[i - 2][j - 2] + 1)

    return matrix[-1][-1]
```

3.3 Тестирование ПО

В таблице 3.1 приведены тестовые случаи для алгоритмов поиска редакционного расстояния.

Таблица 3.1 – Тестовые случаи

№	Строка 1	Строка 2	Левенштейн	Д.-Левенштейн
1	кит	кот	1	1
2	рома	роам	2	1
3	чикипики	чикипик	1	1
4	сальто	сльто	1	1
5		kekw	4	4

3.4 Вывод

В данном разделе были представлены выбор языка программирования, листинги реализаций алгоритмов и результаты тестирования.

4. Исследовательский раздел

В данном разделе будут представлены замеры времени работы реализаций алгоритмов.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование представлены далее.

- Операционная система: macOS Catalina версия 10.15.6;
- Память: 8 GB 1600 MHz DDR3;
- Процессор: 1,8 GHz 2-ядерный процессор Intel Core i5.

Тестирование проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения рабочего стола.

4.2 Сравнительный анализ на основе замеров времени работы алгоритмов

Был проведен замер времени работы каждого из алгоритмов с помощью функции process_time(), которая находится в модуле time языка Python. Замеры времени усреднялись, для каждого алгоритма проводилось по 20 итераций.

Таблица 4.1 – Результаты замеров времени

Длина	Рек. Лев.	Кэш. Лев	Рек. Д.-Лев.	Кэш. Д.-Лев
1	2.72E-06	5.10E-06	2.44E-06	9.83E-06
2	1.17E-05	8.75E-06	1.25E-05	1.56E-05
3	5.53E-05	1.43E-05	7.31E-05	2.88E-05
4	2.65E-04	2.45E-05	2.93E-04	4.04E-05
5	1.51E-03	4.60E-05	1.74E-03	5.23E-05
6	8.55E-03	4.91E-05	8.93E-03	5.03E-05
7	4.35E-02	5.51E-05	4.77E-02	6.28E-05
8	2.43E-01	7.07E-05	2.63E-01	8.40E-05
9	1.36E+00	9.78E-05	1.46E+00	1.21E-04
10	-	1.09E-04	-	1.30E-04
20	-	4.82E-04	-	5.82E-04
30	-	9.28E-04	-	1.33E-03
50	-	2.49E-03	-	3.87E-03
100	-	1.02E-02	-	1.21E-02
200	-	3.98E-02	-	4.93E-02

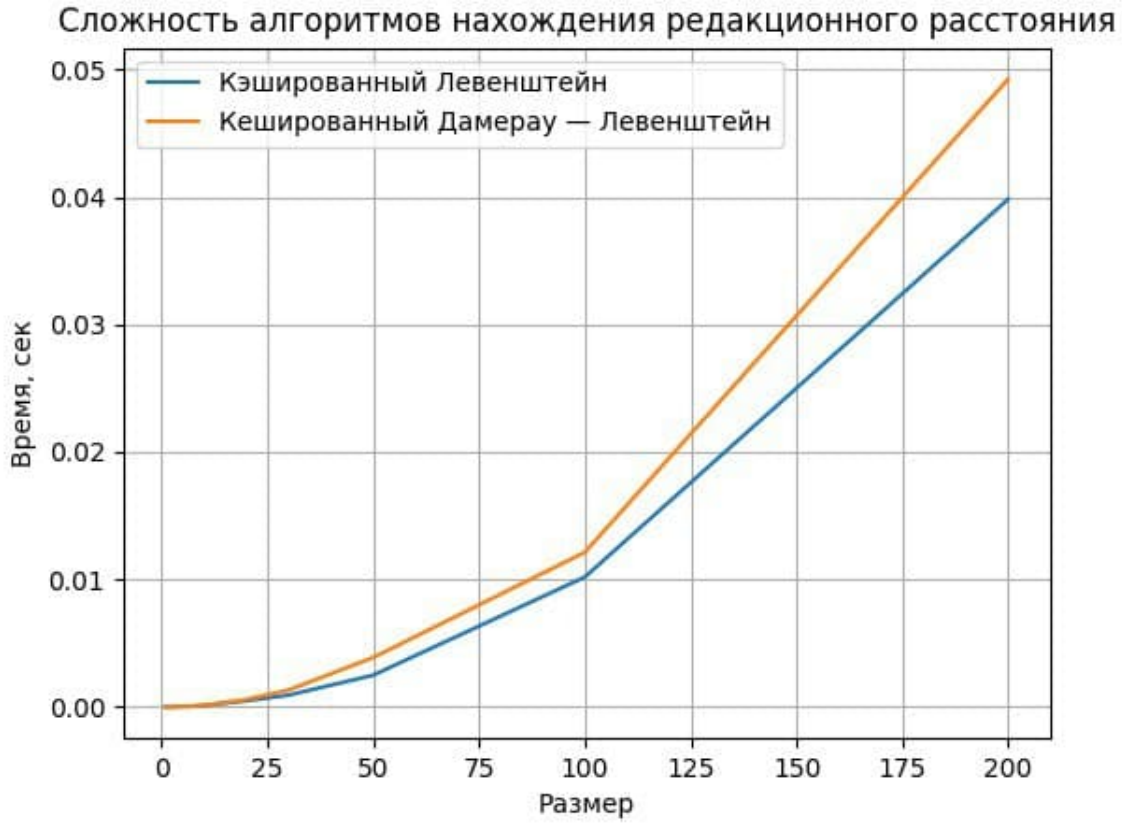

Рисунок 4.1 – Сравнение времени работы кеширующих алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна

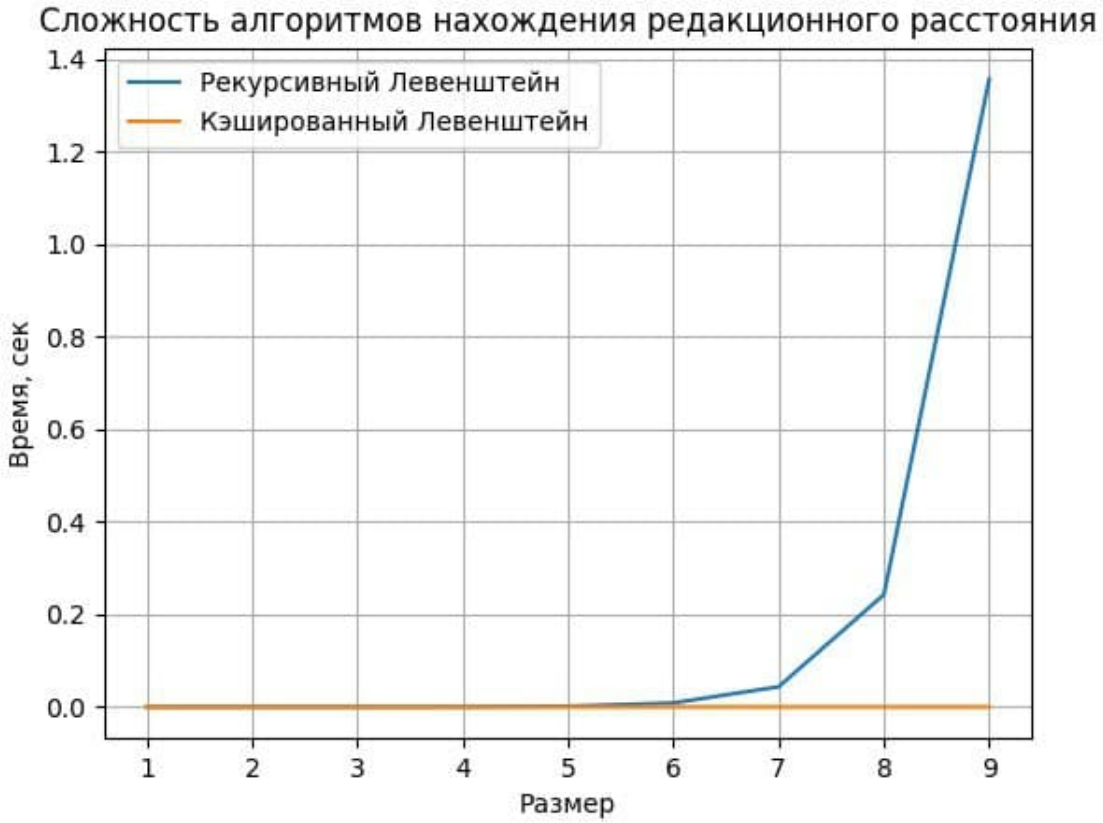

Рисунок 4.2 – Сравнение времени работы рекурсивной и кеширующей реализаций алгоритма Левенштейна

4.3 Вывод

В данном разделе было произведено сравнение количества затраченного времени вышеизложенных алгоритмов.

Исходя из полученных результатов, можно сделать вывод, что при длинах строк менее 30 символов разница по времени между кеширующими реализациями незначительна, однако при увеличении длины строки алгоритм поиска расстояния Левенштейна оказывается быстрее на 20% (при длинах строк равных 200). Это обосновывается тем, что у алгоритма поиска расстояния Дамерау-Левенштейна задействуется дополнительная операция, которая замедляет алгоритм. Также можно сделать вывод, что рекурсивный алгоритм становится менее эффективным, чем кеширующий. Из этого можно сделать вывод о том, что при малых длинах строк (1–5 символов) предпочтительнее использовать рекурсивные алгоритмы, одна-

ко при обработке более длинных строк (болеее 5 символов) кеширующие алгоритмы оказываются многократно более эффективными и рекомендуются к использованию.

Заключение

В ходе выполнения лабораторной работы были решены следующие задачи:

- Изучены алгоритм поиска редакционного расстояния Левенштейна и Дамерау-Левенштейна;
- Получены и применены навыки динамического программирования для некоторых реализаций;
- Реализованы алгоритмы:
  - рекурсивный Левенштейн;
  - кеширующий Левенштейн;
  - рекурсивный Дамерау-Левенштейн;
  - кеширующий Дамерау-Левенштейна.
- Проведено экспериментальное подтверждение различий алгоритмов по временной характеристике;
- Подготовлен отчет по лабораторной работе.

В результате исследований можно прийти к выводу, что использовать рекурсивные алгоритмы нахождения редакционного расстояния следует только на коротких словах, а кеширующие на более длинных.

Литература

1. Welcome to Python [Электронный ресурс] Режим доступа: https://www.python.org/, (дата обращения: 03.10.2021)

2. time - Time access and conversions [Электронный ресурс] Режим доступа: https://docs.python.org/3/library/time.html, (дата обращения: 03.11.2021)

3. Расстояние Левенштейна [Электронный ресурс] Режим доступа: https://habr.com/ru/post/117063/, (дата обращения: 03.11.2021)

4. Расстояние Дамерау-Левенштейна [Электронный ресурс] Режим доступа: https://habr.com/ru/post/114997/, (дата обращения: 03.11.2021)

5. Метрика Левенштейна/Дамерау-Левенштейна [Электронный ресурс] Режим доступа: https://newtechaudit.ru/nechetkoe-sravnenie-strok/, (дата обращения: 03.11.2021)