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 Расстяние Левенштейна и Дамерау-Левенштейна

Студент Гурова Н.А.

Группа ИУ7-54Б

Оценка (баллы) ________________

Преподаватель Волкова Л.Л., Строганов Ю.В.

Москва — 2023 г.

Оглавление

Введение

Целью данной лабораторной работы является изучение, реализация и исследование алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна.

Расстояние Левенштейна – метрика, измеряющая по модулю разность между двумя последовательностями символов. Она определяется как минимальное количество односимвольных операций (вставки, удаления, замены), необходимых для превращения одной последовательности символов в другую.

Расстояния Левенштейна и Дамерау-Левенштейна широко применяются для решения задач компьютерной лингвистики (исправление ошибок в слове, автоматическое распознавание отсканированного текста или речи), биоинформатики (для сравнения генов, хромосом) и других.

Расстояние Дамерау-Левенштейна – модификация расстояния Левештейна. Это мера разницы двух строк символов, определяемая как минимальное количество операций вставки, удаления, замены и транспозиции (перестановки двух соседних символов), необходимых для перевода одной строки в другую.

В рамках выполнения лабораторной работы необходимо решить следующие задачи:

- Изучить расстояния Левенштена и Дамерау-Левенштейна;
- Построить схемы алгоритмов следующих методов: нерекурсивный метод поиска расстояния Левенштейна, нерекурсивный метод поиска Дамерау-Левенштейна, рекурсивный метод поиска Дамерау-Левенштейна, рекурсивный с кешированием метод поиска Дамерау-Левенштейна;
- Создать ПО, реализующее перечисленные выше алгоритмы;
- Сравнить алгоритмы определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);
- Описать и обосновать полученные результаты в отчете о выполненной лабораторной работе;

1 Аналитическая часть

В этом разделе будут представлены описания алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна и их практическое применение.

1.1 Нерекурсивный алгоритм нахождения расстояния Левенштейна

Расстояние Левенштейна [?] между двумя строками – это минимальное количество операций вставки, удаления и замены, необходимых для превращения одной строки в другую.

Каждая операция имеет свою цену (штраф). Редакционным предписанием называется последовательность действий, необходимых для получения из первой строки второй, и минимизирующих суммарную цену. Суммарная цена есть искомое расстояние Левенштейна.

Введем следующие обозначения операций:

- D (англ. delete) – удаление ($w(a, \lambda) = 1$);
- I (англ. insert) – вставка ($w(\lambda, b) = 1$);
- R (англ. replace) – замена ($w(a, b) = 1,\ a \neq b$);
- M (англ. match) - совпадение ($w(a, a) = 0$).

Пусть S_1 и S_2 – две строки (длиной M и N соответственно) над некоторым алфавитом, тогда расстояние Левенштейна можно подсчитать по рекуррентной формуле 1.1.

$$D(i,j) = \begin{cases} 0 & \text{, если i = 0, j = 0,} \\ i & \text{, если j = 0, i > 0,} \\ j & \text{, если i = 0, j > 0,} \\ \min\{ & \\ \quad D(i, j-1) + 1 & \\ \quad D(i-1, j) + 1 & \\ \quad D(i-1, j-1) + m(a[i], b[j]) & \\ \} & \text{, если i > 0, j > 0} \end{cases} \tag{1.1}$$

Где m определяется следующим образом:

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

Нерекурсивный алгоритм реализует формулу 1.1. Функция D составлена таким образом, что для перевода из строки a в строку b требуется выполнить последовательно некоторое количество операций (удаление, вставка, замена) в некоторой последовательности. Полагая, что a', b' – строки a и b без последнего символа соответственно, цена преобразования из строки a в строку b может быть выражена как:

1) сумма цены преобразования строки a' в b и цены проведения операции удаления, которая необходима для преобразования a' в a;

2) сумма цены преобразования строки a в b' и цены проведения операции вставки, которая необходима для преобразования b' в b;

3) сумма цены преобразования из a' в b' и операции замены, предполагая, что a и b оканчиваются на разные символы;

4) цена преобразования из a' в b', предполагая, что a и b оканчиваются на один и тот же символ.

Наименьшей ценой преобразования будет минимальное значение приведенных вариантов.

С ростом i, j прямая реализация формулы 1.1 становится малоэффективной по времени исполнения, так как множество промежуточных значения $D(i, j)$ вычисляются не по одному разу. Для решения этой проблемы можно использовать матрицу для хранения соответствующих промежуточных значений.

Матрица размером $(length(S1) + 1)$x$((length(S2) + 1)$, где $length(S)$ – длина строки S. Значение в ячейке $[i, j]$ равно значению $D(S1[1...i], S2[1...j])$.

Вся таблица (за исключением первого столбца и первой строки) заполняется в соответствии с формулой 1.3.

$$A[i][j] = min\begin{cases} A[i-1][j] + 1 \\ A[i][j-1] + 1 \\ A[i-1][j-1] + m(S1[i], S2[j]) \end{cases}. \tag{1.3}$$

Функция m определена как:

$$m(S1[i], S2[j]) = \begin{cases} 0, & \text{если } S1[i-1] = S2[j-1], \\ 1, & \text{иначе} \end{cases}. \tag{1.4}$$

В результате расстоянием Левенштейна будет ячейка матрицы с индексами $i = length(S1)$ и $j = length(S2)$ при учете, что индексы начинаются с 0.

1.2 Нерекурсивный алгоритм поиска Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна [?] – это мера разницы двух строк символов, определяемая как минимальное количество операций вставки, удаления, замены и транспозиции (перестановки двух соседних символов), необходимых для перевода одной строки в другую. 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 найдено по формуле 1.5.

$$d_{a,b}(i,j) = \begin{cases} \max(i,j), \text{ если } \min(i,j) = 0, \\ \min\{ \\ \qquad d_{a,b}(i,j-1)+1, \\ \qquad d_{a,b}(i-1,j)+1, \\ \qquad d_{a,b}(i-1,j-1)+m(a[i],b[j]), \text{ иначе} \\ \qquad \left[\begin{array}{c} d_{a,b}(i-2,j-2)+1, \text{ если } i,j>1; \\ a[i]=b[j-1]; \\ b[j]=a[i-1] \\ \infty, \text{ иначе} \end{array} \right. \\ \} \end{cases}, \tag{1.5}$$

Формула выводится по тем же соображениям, что и формула 1.1.

1.3 Рекурсивный алгоритм поиска Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна может быть найдено по формуле 1.6.

$$d_{a,b}(i,j) = \begin{cases} \max(i,j), \text{ если } \min(i,j) = 0, \\ \min\{ \\ \qquad d_{a,b}(i,j-1)+1, \\ \qquad d_{a,b}(i-1,j)+1, \\ \qquad d_{a,b}(i-1,j-1)+m(a[i],b[j]), \text{ иначе} \\ \qquad \left[\begin{array}{c} d_{a,b}(i-2,j-2)+1, \text{ если } i,j>1; \\ a[i]=b[j-1]; \\ b[j]=a[i-1] \\ \infty, \text{ иначе} \end{array} \right. \\ \} \end{cases}, \tag{1.6}$$

1.4 Рекурсивный с кешированием алгоритм поиска Дамерау-Левенштейна

Рекурсивный алгоритм заполнения можно оптимизировать по времени выполнения с использованием кеша. В качестве кеша используется матрица. Суть данной оптимизации заключается в параллельном заполнении матрицы при выполнении рекурсии. В случае, если рекурсивный алгоритм выполняет прогон для данных, которые еще не были обработаны, результат нахождения расстояния заносится в матрицу. В случае, если обработанные ранее данные встречаются снова, для них расстояние не находится и алгоритм переходит к следующему шагу.

Вывод

В данном разделе были рассмотрены алгоритмы поиска расстояния Левенштейна и расстояния Дамерау-Левенштейна. В частности были приведены рекурентные формулы работы алгоритмов, объяснена разница между расстоянием Левенштейна и расстоянием Дамерау-Левенштейна.

2 Конструкторская часть

В этом разделе будут приведены требования к вводу и программе, а также схемы алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна.

2.1 Требования к вводу

На вход подаются две строки, причем буквы верхнего и нижнего регистров считаются различными.

2.2 Требования к программе

При вводе двух пустых строк программа не должна аварийно завершатся. Вывод программы - число (расстояние Левенштейна или Дамерау-Левенштейна)

2.3 Разработка алгоритма поиска расстояния Левенштейна

На рисунке 2.1 приведена схема нерекурсивного алгоритма нахождения расстояния Левенштейна.

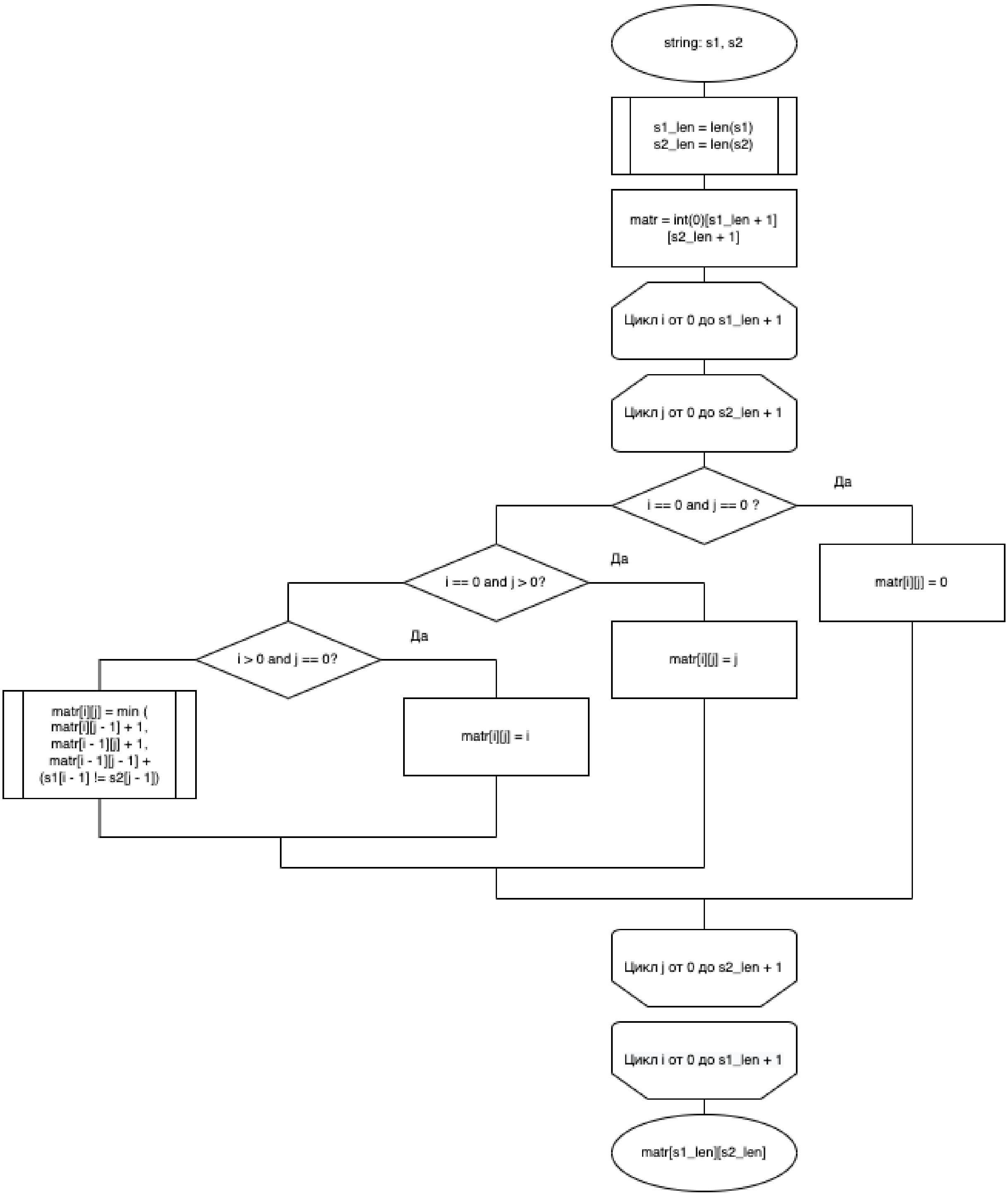

Рисунок 2.1 – Схема нерекурсивного алгоритма нахождения расстояния Левенштейна

2.4 Разработка алгоритма поиска расстояния Дамерау-Левенштейна

На рисунке 2.2 приведена схема нерекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна.

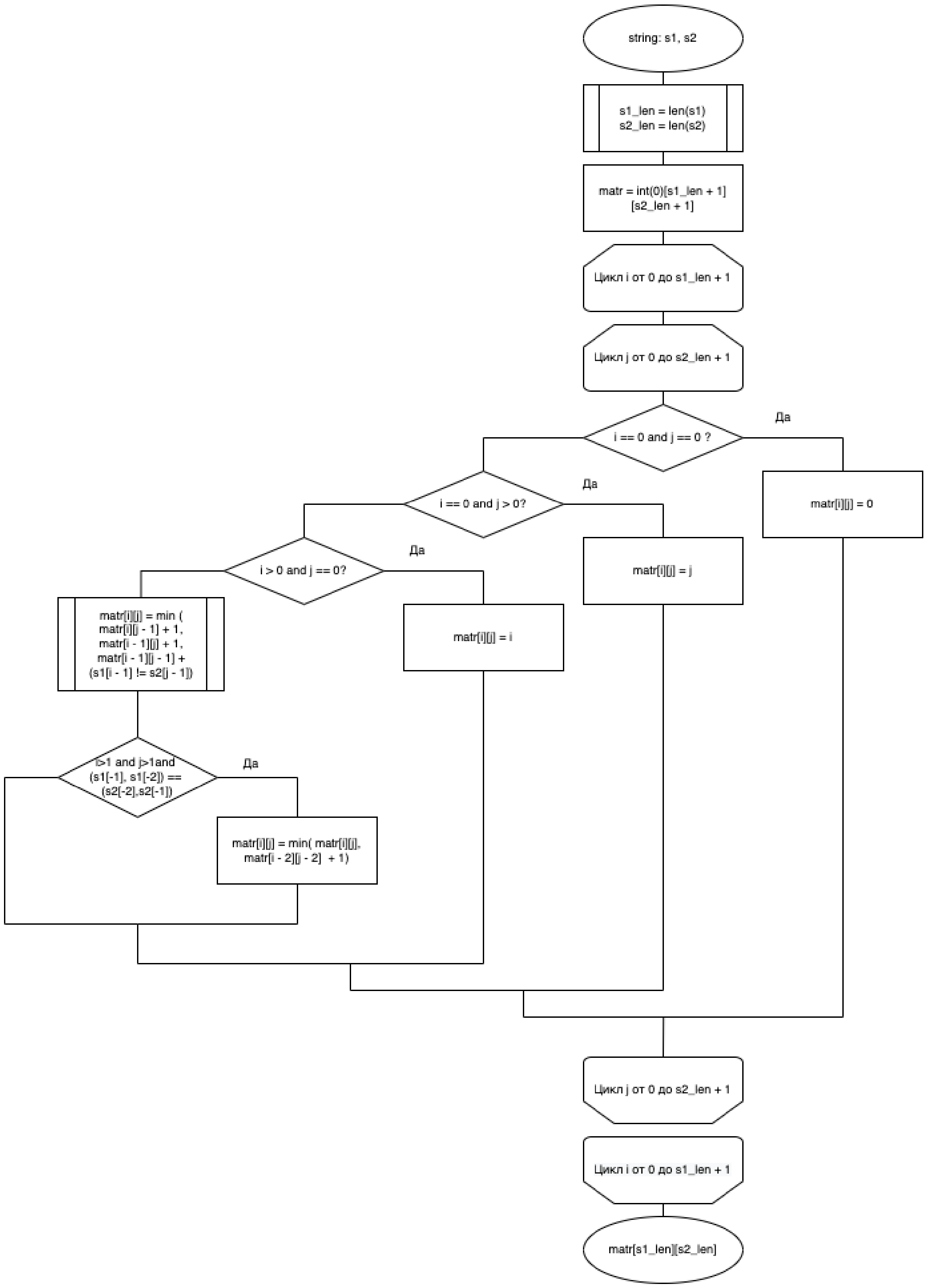

Рисунок 2.2 – Схема нерекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

На рисунке 2.3 приведена схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна.

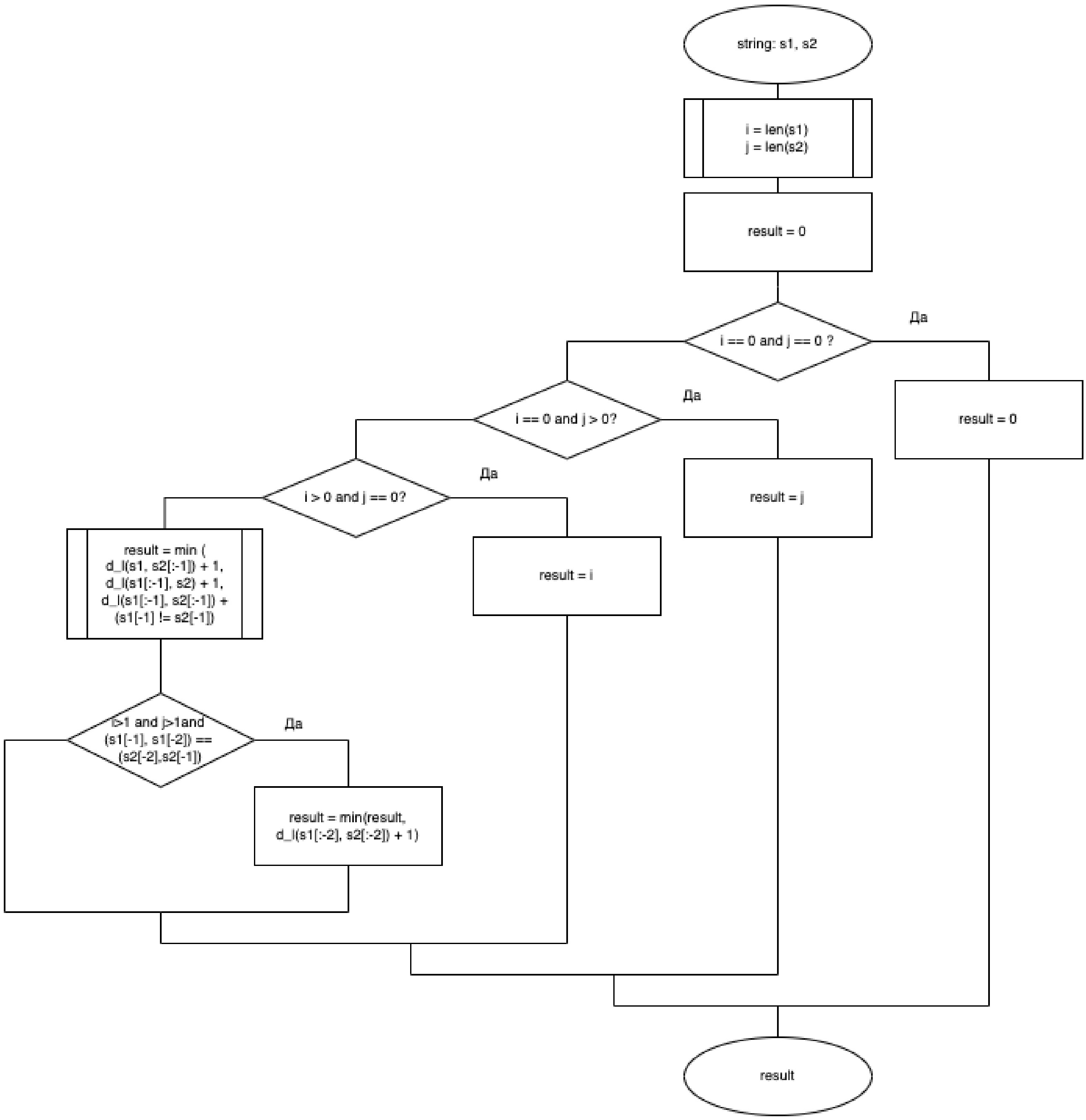

Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

На рисунке 2.4 приведена схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с использованием кеша в виде матрицы.

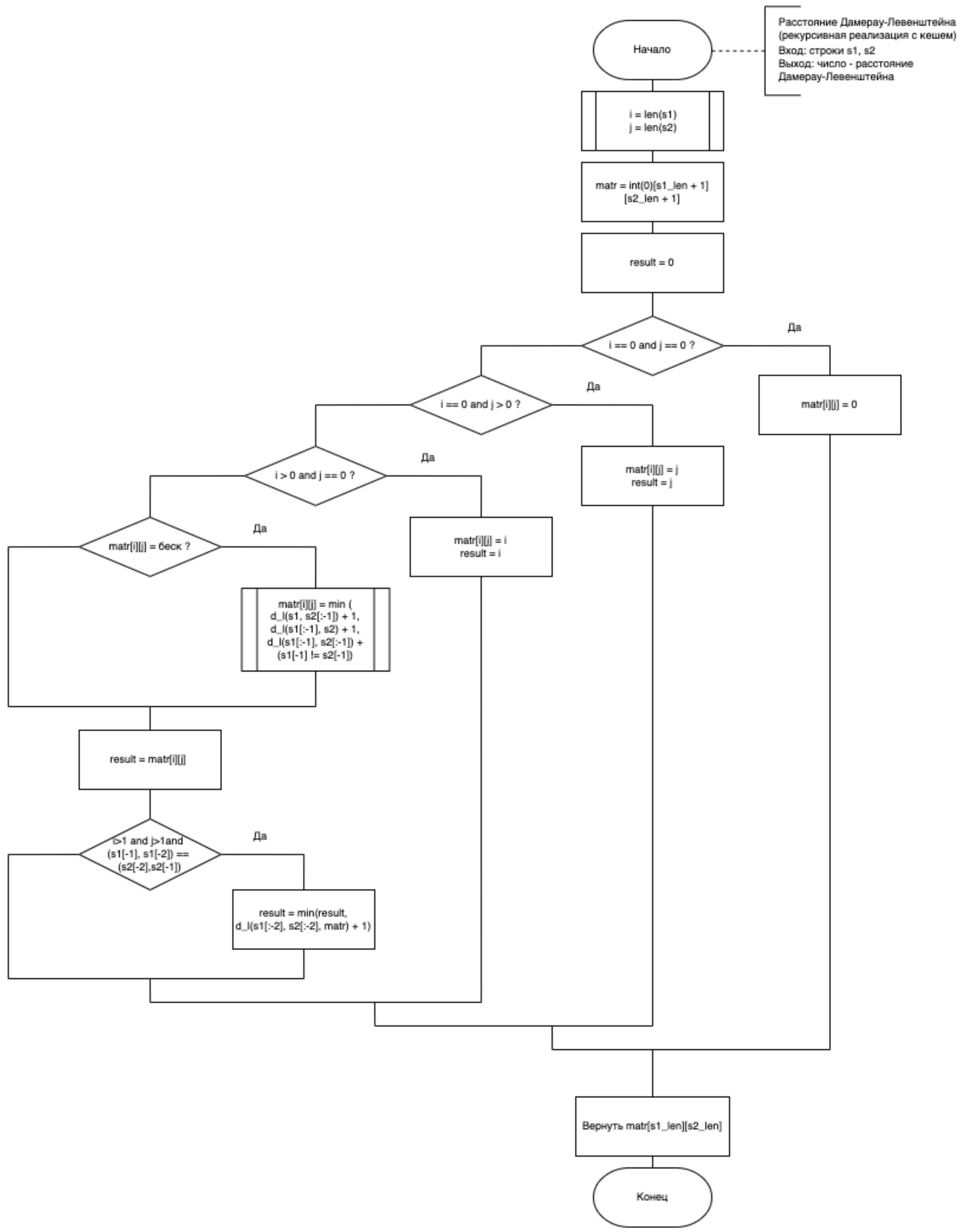

Рисунок 2.4 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с использованием кеша в виде матрицы

Вывод

Перечислены требования к вводу и программе, а также на основе теоретических данных, полученных из аналитического раздела, были построены схемы требуемых алгоритмов.

3 Технологическая часть

В данном разделе будут приведены требования к программному обеспечению, средства реализации и листинги кода.

3.1 Требования к ПО

Программа принимает две строки (регистрозависимые). В качестве результата возвращается число, равное редакторскому расстоянию. Необходимо реализовать возможность подсчета процессорного времени и пиковой использованной памяти для каждого из алгоритмов.

3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран ЯП Python [?].

Данный язык имеет все небходимые инструменты для решения поставленной задачи.

Время работы алгоритмов было замерено с помощью функции time() из библиотеки time [?].

3.3 Сведения о модулях программы

Программа состоит из четырех модулей:

1. algorithms.py - хранит реализацию алгоритмов;
2. unit_tests.py - хранит реализацию тестирующей системы и тесты;
3. time_memory.py - хранит реализацию системы замера памяти и времени;
4. tools.py - хранит реализацию вспомогательных функций.

3.4 Листинг кода

В листингах 3.1, 3.2, 3.3, 3.4 приведены реализации алгоритмов нахождения расстояния Левенштейна и Дамерау–Левенштейна.

Листинг 3.1 – Функция нахождения расстояния Левенштейна нерекурсивным методом.

```
def non_recursive_levenshtein(source: str, target: str) -> int:
    source_len = len(source)
    target_len = len(target)

    matrix_dist = [[0 for i in range(target_len + 1)] for j in range(source_len + 1)]

    for i in range(source_len + 1):
        for j in range(target_len + 1):
            if i == 0 and j == 0:
                matrix_dist[i][j] = 0
            elif i == 0 and j > 0:
                matrix_dist[i][j] = j
            elif i > 0 and j == 0:
                matrix_dist[i][j] = i
            else:
                matrix_dist[i][j] = min(matrix_dist[i][j - 1] + 1,
                                        matrix_dist[i - 1][j] + 1,
                                        matrix_dist[i - 1][j - 1] + (source[i - 1] != target[j - 1]))
    return matrix_dist[source_len][target_len]
```

Листинг 3.2 – Функция нахождения расстояния Дамерау–Левенштейна нерекурсивным методом.

```
def non_recursive_damerau_levenshtein(source: str, target: str) -> int:
    source_len = len(source)
    target_len = len(target)

    matrix_dist = [[0 for i in range(target_len + 1)] for j in range(source_len + 1)]

```

```
    for i in range(source_len + 1):
        for j in range(target_len + 1):
            if i == 0 and j == 0:
                matrix_dist[i][j] = 0
            elif i == 0 and j > 0:
                matrix_dist[i][j] = j
            elif i > 0 and j == 0:
                matrix_dist[i][j] = i
            else:
                matrix_dist[i][j] = min(matrix_dist[i][j - 1] + 1,
                                        matrix_dist[i - 1][j] + 1,
                                        matrix_dist[i - 1][j - 1]
                                          + (source[i - 1] !=
                                          target[j - 1]))

                if i > 1 and j > 1 and source[i - 1] == target[j
                  - 2] and source[i - 2] == target[j - 1]:
                    exchange_dist = matrix_dist[i - 2][j - 2] + 1
                    matrix_dist[i][j] = min(matrix_dist[i][j],
                      exchange_dist)

    return matrix_dist[source_len][target_len]
```

Листинг 3.3 – Функция нахождения расстояния Дамерау–Левенштейна с использованием рекурсии.

```
def _rdl_wrap(source: str, target: str, i: int, j: int) -> int:
    answer = -1

    if i == 0 and j == 0:
        answer = 0
    elif i == 0:
        answer = j
    elif j == 0:
        answer = i
    else:

        left = _rdl_wrap(source, target, i - 1, j) + 1
        up = _rdl_wrap(source, target, i, j - 1) + 1
        left_up = _rdl_wrap(source, target, i - 1, j - 1) +
          (source[i - 1] != target[j - 1])

```

```
        answer = min(left, up, left_up)

        if i > 1 and j > 1 and source[i - 1] == target[j - 2] and source[i - 2] == target[j - 1]:
            exchange_dist = _rdl_wrap(source, target, i - 2, j - 2) + 1
            answer = min(answer, exchange_dist)

    return answer


def recursive_damerau_levenshtein(source: str, target: str) -> int:
    source_len = len(source)
    target_len = len(target)

    return _rdl_wrap(source, target, source_len, target_len)
```

Листинг 3.4 – Функция нахождения расстояния Дамерау–Левенштейна рекурсивным методом с использованием кеша.

```
def _rdl_cache_wrap(source: str, target: str, i: int, j: int, cache: list[list[int]]) -> int:
    if cache[j][i] != -1:
        return cache[j][i]

    if i == 0 and j == 0:
        cache[j][i] = 0
        return cache[j][i]
    elif i == 0 and j > 0:
        cache[j][i] = j
        return cache[j][i]
    elif j == 0 and i > 0:
        cache[j][i] = i
        return cache[j][i]
    else:
        left = _rdl_cache_wrap(source, target, i - 1, j, cache) + 1
        up = _rdl_cache_wrap(source, target, i, j - 1, cache) + 1
        left_up = _rdl_cache_wrap(source, target, i - 1, j - 1, cache) + (source[i - 1] != target[j - 1])

```

```python
        cache[j][i] = min(left, up, left_up)

        if i > 1 and j > 1 and source[i - 1] == target[j - 2] and source[i - 2] == target[j - 1]:
            exchange = _rdl_cache_wrap(source, target, i - 2, j - 2, cache) + 1
            cache[j][i] = min(cache[j][i], exchange)

    return cache[j][i]


def recursive_damerau_levenshtein_with_cache(source: str, target: str) -> int:
    source_len = len(source)
    target_len = len(target)

    cache = [[-1 for i in range(source_len + 1)] for j in range(target_len + 1)]
    return _rdl_cache_wrap(source, target, source_len, target_len, cache)
```

3.5 Функциональные тесты

В таблице 3.1 приведены функциональные тесты для алгоритмов вычисления расстояния Левенштейна (в таблице столбец подписан "Левенштейн") и Дамерау-Левенштейна (в таблице - "Дамерау-Л."). Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты

			Ожидаемый результат	
№	Строка 1	Строка 2	Левенштейн	Дамерау-Л.
1	"пустая строка"	"пустая строка"	0	0
2	"пустая строка"	a	1	1
3	b	"пустая строка"	1	1
4	asdfv	"пустая строка"	5	5
5	"пустая строка"	sdfvs	5	5
8	lll	lll	0	0
9	qwem	qwem	0	0
10	aa	cg	2	2
11	kot	sobaka	5	5
12	stroka	sobaka	3	3
13	kot	kod	1	1
14	cat	caaat	2	2
15	cat	catty	2	2
16	cat	tac	2	2
17	recur	norecur	2	2
18	1234	2143	3	2
19	mriak	mriakmriak	5	5
20	aaaaa	aa	3	3

Вывод

Были разработаны и протестированы алгоритмы: нахождения расстояния Левенштейна нерекурсивно, нахождения расстояния Дамерау – Левенштейна нерекурсивно, рекурсивно, а также рекурсивно с кешированием.

4 Исследовательская часть

В данном разделе приводятся результаты замеров алгоритмов по пиковой памяти и процессорному времени.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- Операционная система: macOS Monterey 12.5.1
- Память: 16 Гб.
- Процессор: 2,3 ГГц 4-ядерный процессор Intel Core i5.

Во время тестирования устройство было подключено к сети электропитания, нагружено приложениями окружения и самой системой тестирования.

4.2 Время выполнения алгоритмов

Алгоритмы тестировались при помощи функции time() из библиотеки time языка Python. Данная функция возвращает количество секунд, прошедших с начала эпохи, типа float.

Контрольная точка возвращаемого значения не определна, поэтому допустима только разница между результатами последовательных вызовов.

Замеры времени для каждой длины слов проводились 1000 раз. В качестве результата взято среднее время работы алгоритма на данной длине слова. При каждом запуске алгоритма, на вход подавались случайно сгенерированные строки. Тестовые пакеты создавались до начала замера времени.

На рисунке 2.2 приведена схема нерекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна.

Результаты замеров приведены на рисунке 4.1 (в микросекундах).

TIME RESULT (microseconds)

Algorithm \ Length	(0, 0)	(1, 1)	(3, 3)	(5, 5)	(10, 10)	(25, 25)	(50, 50)
Levenshtein	2.49	3.38	10.76	24.51	79.71	461.63	1750.75
Damerau-Levenshtein	2.19	3.45	11.93	27.00	93.65	587.36	2078.44
Recursive Damerau-Levenshtein	0.40	1.20	30.68	818.51	-	-	-
Recursive Damerau-Levenshtein with cache	1.54	3.08	15.12	36.76	135.75	-	-

Рисунок 4.1 – Результаты замеров времени (в микросекундах)

4.3 Использование памяти

Алгоритмы тестировались при помощи функции get_traced_memory() из библиотеки tracemalloc языка Python, которая возвращает пиковый количество памяти, использованное процессором, на определенном этапе выполнения программы.

При каждом запуске алгоритма, на вход подавались случайно сгенерированные строки. Тестовые пакеты создавались до начала замера памяти.

Результаты замеров приведены на рисунке 4.2 (в байтах).

MEMORY RESULT (bytes)

Algorithm \ Length	(0, 0)	(1, 1)	(3, 3)	(5, 5)	(10, 10)	(25, 25)	(50, 50)
Levenshtein	440.000	520.000	584.000	912.000	2168.000	8104.000	23384.000
Damerau-Levenshtein	440.000	520.000	584.000	856.000	1944.000	7320.000	22040.000
Recursive Damerau-Levenshtein	0.000	552.000	2064.000	2064.000	-	-	-
Recursive Damerau-Levenshtein with cache	104.000	520.000	584.000	856.000	6664.000	-	-

Рисунок 4.2 – Результаты замеров памяти (в байтах)

Вывод

В результате замеров можно прийти к выводу, что матричная реализация алгоритмов нахождения расстояний заметно выигрывает по времени при росте строк, но проигрывает по количеству затрачиваемой памяти.

Заключение

В ходе выполнения лабораторной работы были решены следующие задачи:

- изучены алгоритмы нахождения расстояний Левенштейна и Дамерау-Левенштейна;
- реализованы алгоритмы поиска расстояния Левенштейна и расстояния Дамерау-Левенштейна без рекурсии;
- реализованы рекурсивные алгоритмы поиска расстояния Дамерау-Левенштейна с и без матрицы-кеша;
- проведен сравнительный анализ линейной и рекурсивной реализаций алгоритмов определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);
- подготовлен отчет о лабораторной работе.

Экспериментально было подтверждено различие во временной эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализаций на различных длин строк. Было получено, что рекурсивная реализация алгоритмов без кеширования в 3-4 раза проигрывает по памяти нерекурсивной реализации. Однако ее можно улучшить, добавив кеширование, и получить небольшой (1.5 раза) выигрыш по памяти по сравнению с нерекурсивными алгоритмами. Анализ временных затрат показал, что для длинных строк (10 символов и больше) рекурсивная реализация алгоритмов работает в 1.5 раза дольше, чем нерекурсивная.