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 Расстояние Левенштейна и Дамерау–Левенштейна

Студент Жаворонкова А. А.

Группа ИУ7-56Б

Оценка (баллы)

Преподаватель Волкова Л. Л.

Москва — 2023 г.

Содержание

Введение

Существует несколько важных задач, для решения которых нужны алгоритмы сравнения строк. Об этих алгоритмах и пойдет речь в данной работе. Подобные алгоритмы используются при [4]:

— исправлении ошибок в тексте, предлагая заменить введенное слово с ошибкой на наиболее подходящее;

— поиске слова в тексте по подстроке;

— сравнении целых текстовых файлов.

Целью данной работы исследование алгоритмов нахождения расстояний Левенштейна и Дамерау — Левенштейна. Для достижения поставленной цели необходимо выполнить следующие задачи:

— описать расстояния Левенштейна и Дамерау — Левенштейна;

— реализовать алгоритмы нахождения расстояний Левенштейна и Дамерау — Левенштейна;

— провести тестирование по методу черного ящика для реализаций алгоритмов нахождения расстояний Левенштейна и Дамерау — Левенштейна;

— провести сравнительный анализ по времени рекурсивной и матричной реализации алгоритма нахождения расстояния Левенштейна;

— провести сравнительный анализ по времени матричной и с кешем реализации алгоритма нахождения расстояния Левенштейна;

— провести сравнительный анализ по времени алгоритмов нахождения расстояния Левенштейна и Дамерау — Левенштейна;

— описать и обосновать полученные результаты в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

1 Аналитическая часть

В данном разделе будут рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау — Левенштейна.

1.1 Расстояние Левенштейна

Расстояние Левенштейна между двумя строками — минимальное количество редакционных операций вставки, удаления, замены, необходимых для превращения одной строки в другую [4]. Были введены следующие обозначения:

— I (англ. Insert) — вставка символа в произвольной позиции;

— D (англ. Delete) — удаление символа в произвольной позиции;

— R (англ. Replace) — замена символа на другой;

— M (англ. Match) — совпадение двух символов.

Пусть исходная строка — S_1, целевая — S_2. $S_1[1 \dots i]$ - подстрока S_1 длиной i символов, начиная с начального. $S_2[1 \dots j]$ - подстрока S_2 длиной j символов, начиная с начального. Пусть L_1 — длина строки S_1, L_2 — длина строки S_2.

С учетом введенных обозначений, расстояние Левенштейна было подсчитано по следующей рекуррентной формуле:

$$D(S_1[1\dots i], S_2[1\dots j]) = \begin{cases} 0, i = 0, j = 0 \\ i, j = 0, i > 0 \\ j, i = 0, j > 0 \\ \min\{ \\ \quad D(S_1[1\dots i], S_2[1\dots j-1]) + 1 \\ \quad D(S_1[1\dots i-1], S_2[1\dots j]) + 1, \\ \quad D(S_1[1\dots i-1], S_2[1\dots j-1]) + \\ \quad + \begin{cases} 0, S_1[i] == S_2[j] \\ 1, \end{cases} \\ \}, i > 0, j > 0 \end{cases} \tag{1.1}$$

1.2 Расстояние Дамерау — Левенштейна

Дамерау ввел четвертую операцию — транспозицию соседних символов со штрафом 1. Обозначение — X (англ. Exchange). Следовательно, расстояние Дамерау — Левенштейна — это минимальное количество редакционных операций вставки, удаления, замены, обмена, необходимых для превращения одной строки в другую.

Расстояние Дамерау — Левенштейна считается по рекурсивной формуле (1.2).

$$D(S_1[1\dots i], S_2[1\dots j]) = \begin{cases} 0, i = 0, j = 0 \\ i, j = 0, i > 0 \\ j, i = 0, j > 0 \\ \min\{ \\ \quad D(S_1[1\dots i], S_2[1\dots j-1]) + 1 \\ \quad D(S_1[1\dots i-1], S_2[1\dots j]) + 1 \\ \quad D(S_1[1\dots i-2], S_2[1\dots j-2]) + 1, \\ \qquad S_1[i] == S_2[j-1], \\ \qquad S_1[i-1] == S_2[j] \\ \quad D(S_1[1\dots i-1], S_2[1\dots j-1]) + \\ \qquad + \begin{cases} 0, S_1[i] == S_2[j] \\ 1, \end{cases} \\ \} \end{cases} \tag{1.2}$$

1.3 Нерекурсивный алгоритм нахождения расстояния Левенштейна

Идея алгоритма: заполнять матрицу не от конца к началу, как при рекурсивных алгоритмах, а наоборот, от начала к концу.

Создается матрица с числом строк $(L_1 + 1)$ и числом столбцов $(L_2 + 1)$. Так как первый столбец и первая строка соответствуют пустым строкам, то в полученной матрице они заполняются тривиально. Затем каждый элемент $A[i; j]$ матрицы вычисляется на основе элементов $A[i-1; j]$, $A[i; j-1]$, $A[i-1; j-1]$ по следующей формуле:

$$
\begin{aligned}
A[i;j] = \min\{ & \\
& A[i-1;j] + 1 \\
& A[i;j-1] + 1 \\
& A[i-1;j-1] + \begin{cases} 0, S_1[i] == S_2[j] \\ 1, \end{cases} \\
& \}, i > 0, j > 0
\end{aligned}
\tag{1.3}
$$

После заполнения всей матрицы результатом будет являться элемент $A[L_1; L_2]$.

Возможной оптимизацией данного алгоритма является минимизация используемой памяти. Поскольку для вычисления текущего элемента матрицы необходимы только 2 строки: текущая и предыдущая, то можно хранить не всю матрицу, а только последние 2 строки.

1.4 Нерекурсивный алгоритм нахождения расстояния Дамерау — Левенштейна

Идея алгоритма аналогична идее нахождения расстояния Левенштейна. Однако в случае нахождения расстояния Дамерау — Левенштейна каждый элемент $A[i;j]$ матрицы вычисляется на основе элементов $A[i-1;j]$, $A[i;j-1]$, $A[i-1;j-1]$, $A[i-2;j-2]$ по следующей формуле:

$$
\begin{aligned}
A[i;j] = \min\{ & \\
& A[i-1;j] + 1 \\
& A[i;j-1] + 1 \\
& A[i-2;j-2] + 1, S_1[i] == S_2[j-1], S_1[i-1] == S_2[j] \\
& A[i-1;j-1] + \begin{cases} 0, S_1[i] == S_2[j] \\ 1, \end{cases} \\
& \}, i > 0, j > 0
\end{aligned}
\tag{1.4}
$$

Аналогично, возможна минимизация используемой памяти: хранение

не всей матрицы, а только последних трех строк.

1.5 Рекурсивный алгоритм нахождения расстояния Дамерау — Левенштейна

Рекурсивный алгоритм вычисления расстояния Дамерау — Левенштейна реализует формулу (1.2).

Основной проблемой данного алгоритма являются повторные вычисления. В качестве оптимизации можно хранить кэш в виде матрицы.

1.6 Рекурсивный алгоритм нахождения расстояния Дамерау — Левенштейна с использованием кэша

В качестве оптимизации рекурсивного алгоритма заполнения можно использовать кэш, который будет представлять собой матрицу. Суть оптимизации — при выполнении рекурсии происходит заполнение матрицы.

Изначально матрица заполнена бесконечно большими числами. Если рекурсивный алгоритм выполняет прогон для данных, которые еще не были обработаны, то результат нахождения заносится в матрицу. Иначе, если обработанные данные встречаются снова, то для них расстояние не находится и алгоритм переходит к следующему шагу.

Вывод

В данном разделе было дано математическое описание расстояний Левенштейна и Дамерау — Левенштейна, которые являются рекуррентными, что позволяет реализовать их как рекурсивно, так и итерационно.

К разрабатываемой программе предъявляются следующие требования:

1. Программа должна предоставлять функциональность расчета расстояния Левенштейна матричным и оптимизированным матричным алгоритмами, а также расстояния Дамерау — Левенштейна рекурсивным, рекурсивным с кэшем, матричным и оптимизированным матричным алгоритмами.

2. Реализуемое ПО будет работать в двух режимах — пользовательском, в котором можно выбрать алгоритм и вывести для него рассчитанное значение, а также экспериментальном режим, в котором можно произвести сравнение реализаций алгоритмов по времени работы на различных входных данных.

3. В первом режиме в качестве входных данных в программу будет подаваться две строки, также реализовано меню для вызова алгоритмов и замеров времени. Ограничением для работы программного продукта является то, что программе на вход может подаваться строка на английском или русском языке, а также программа должна корректно обрабатывать случай ввода пустых строк.

4. Во втором режиме будет происходить измерение процессорного времени работы программы, будут построены зависимости времени расстояний от совпадающих длин входных строк. Входные строки будут сгенерированы автоматически для заданной совпадающей длины строк.

2 Конструкторская часть

В этом разделе будут представлено описание используемых типов данных, а также схемы алгоритмов вычисления расстояний Левенштейна и Дамерау — Левенштейна.

2.1 Сведения о модулях программы

Программа состоит из четырех модулей:

— *main.py* — файл, содержащий весь служебный код;

— *algorithms.py* — файл, содержащий реализации алгоритмов;

— *graph.py* — файл, содержащий код для построения графиков;

— *tests.py* — файл, содержащий модульные тесты.

2.2 Разработка алгоритмов

На рисунках 2.1 — 2.7 представлены схемы алгоритмов вычисления расстояний Левенштейна и Дамерау — Левенштейна.

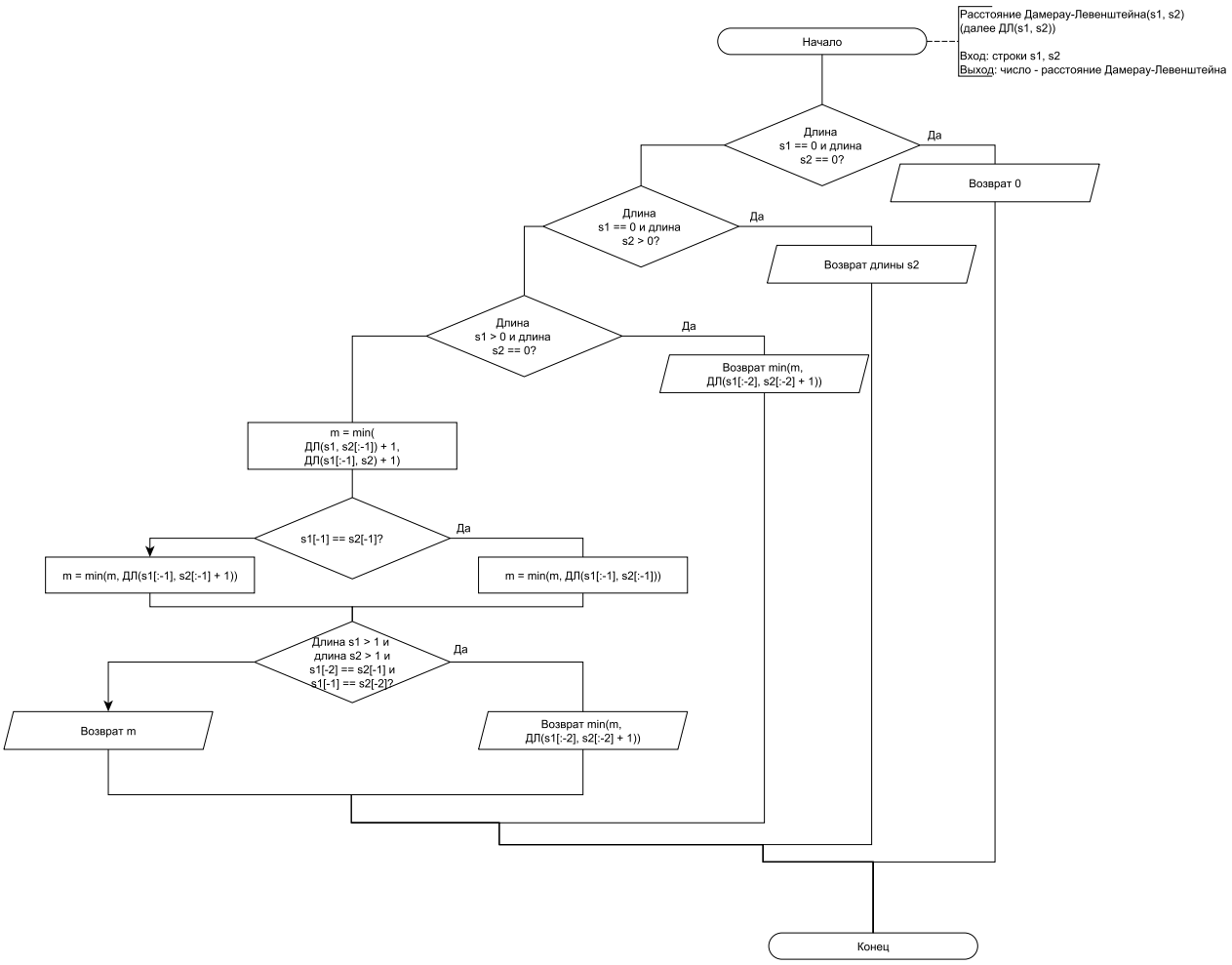

Рисунок 2.1 – Схема рекурсивного алгоритма нахождения расстояния Дамерау — Левенштейна

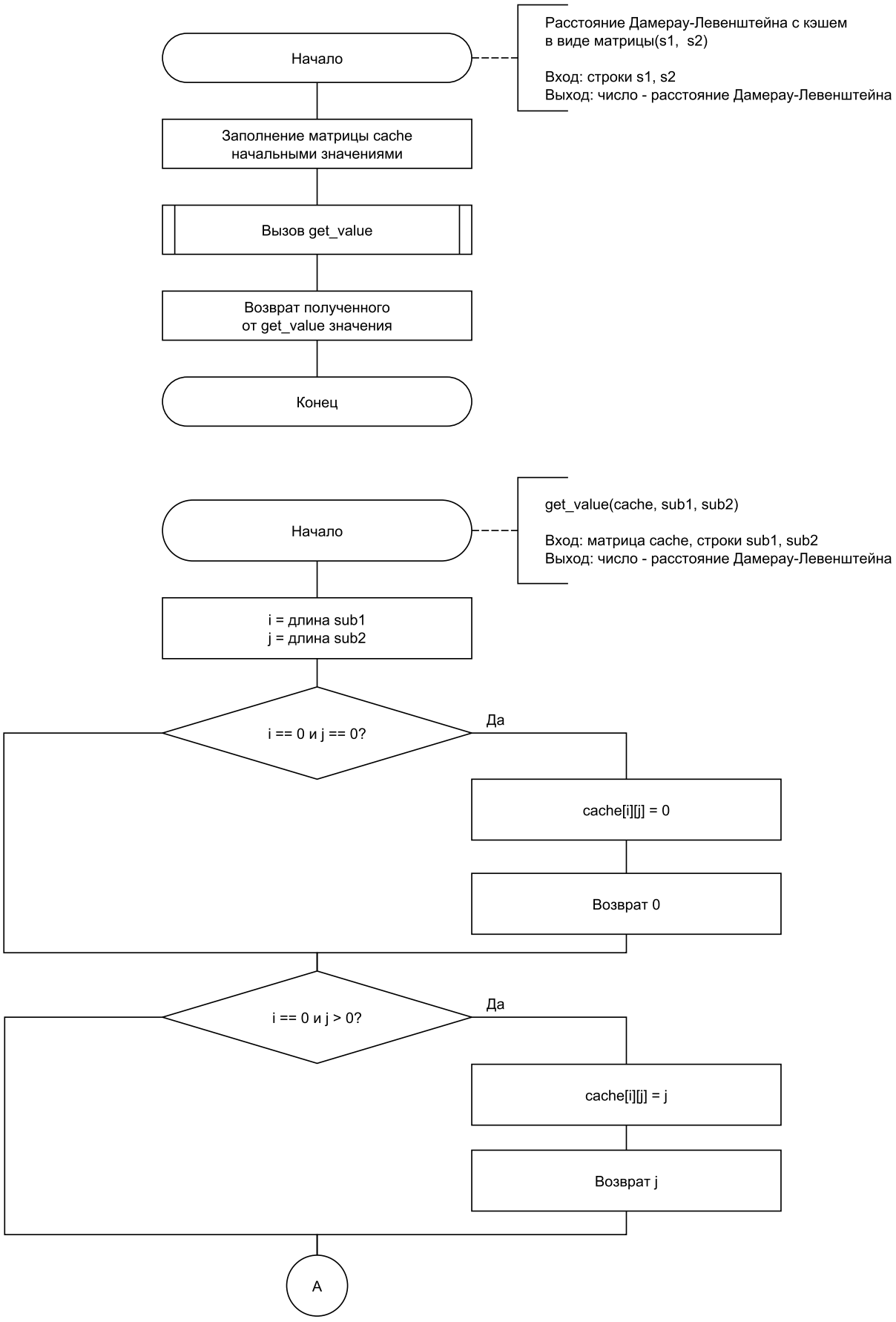

Рисунок 2.2 – Схема рекурсивного алгоритма нахождения расстояния Дамерау — Левенштейна с использованием кэша в виде матрицы (часть 1)

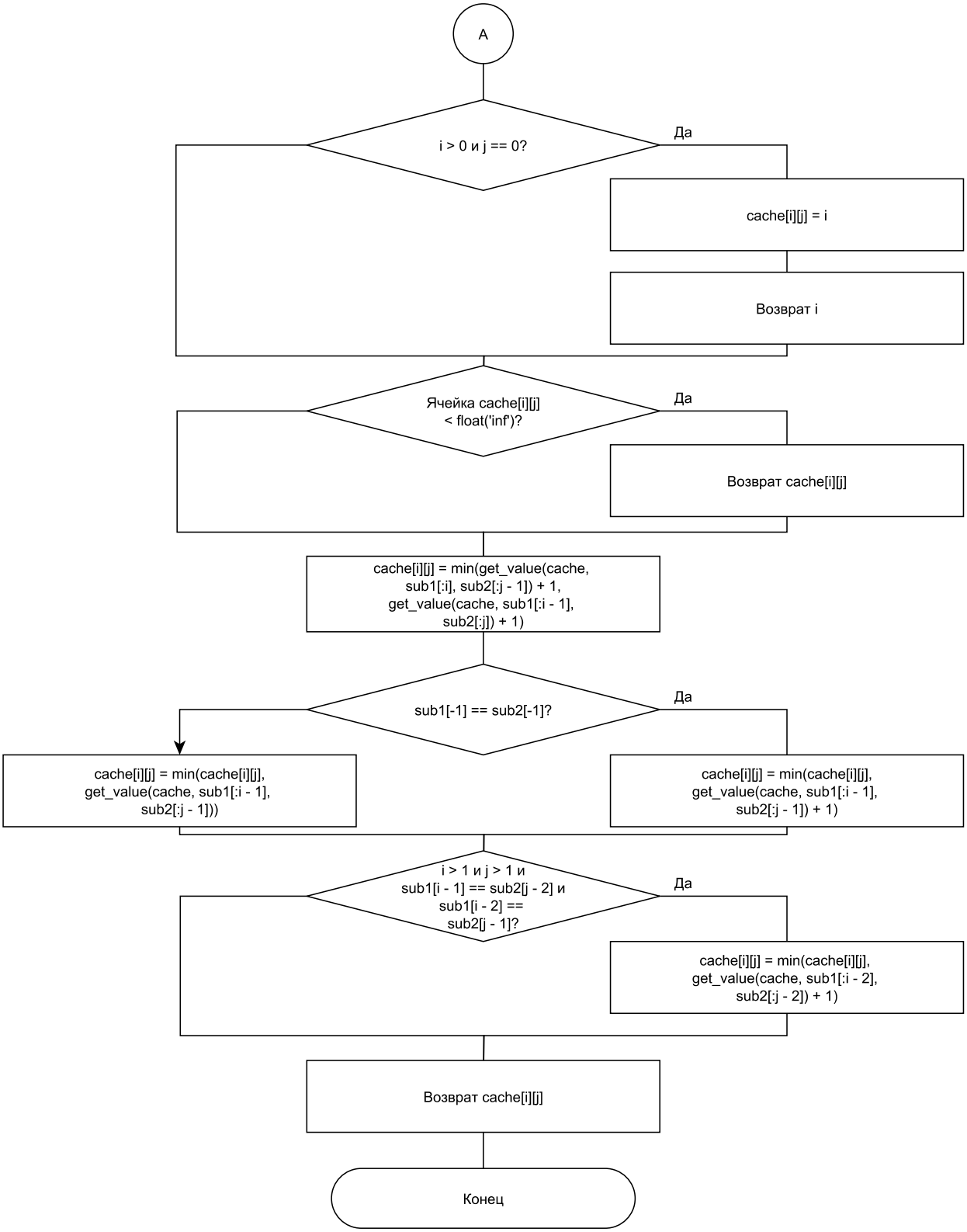

Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Дамерау — Левенштейна с использованием кэша в виде матрицы (часть 2)

Матричные алгоритмы нахождения расстояний Левенштейна и Дамерау — Левенштейна можно описать в виде одной схемы алгоритмов, представленной на рисунках 2.4–2.5. Если $flag == true$, то будет вычислено расстояние

Дамерау — Левенштейна, иначе — расстояние Левенштейна.

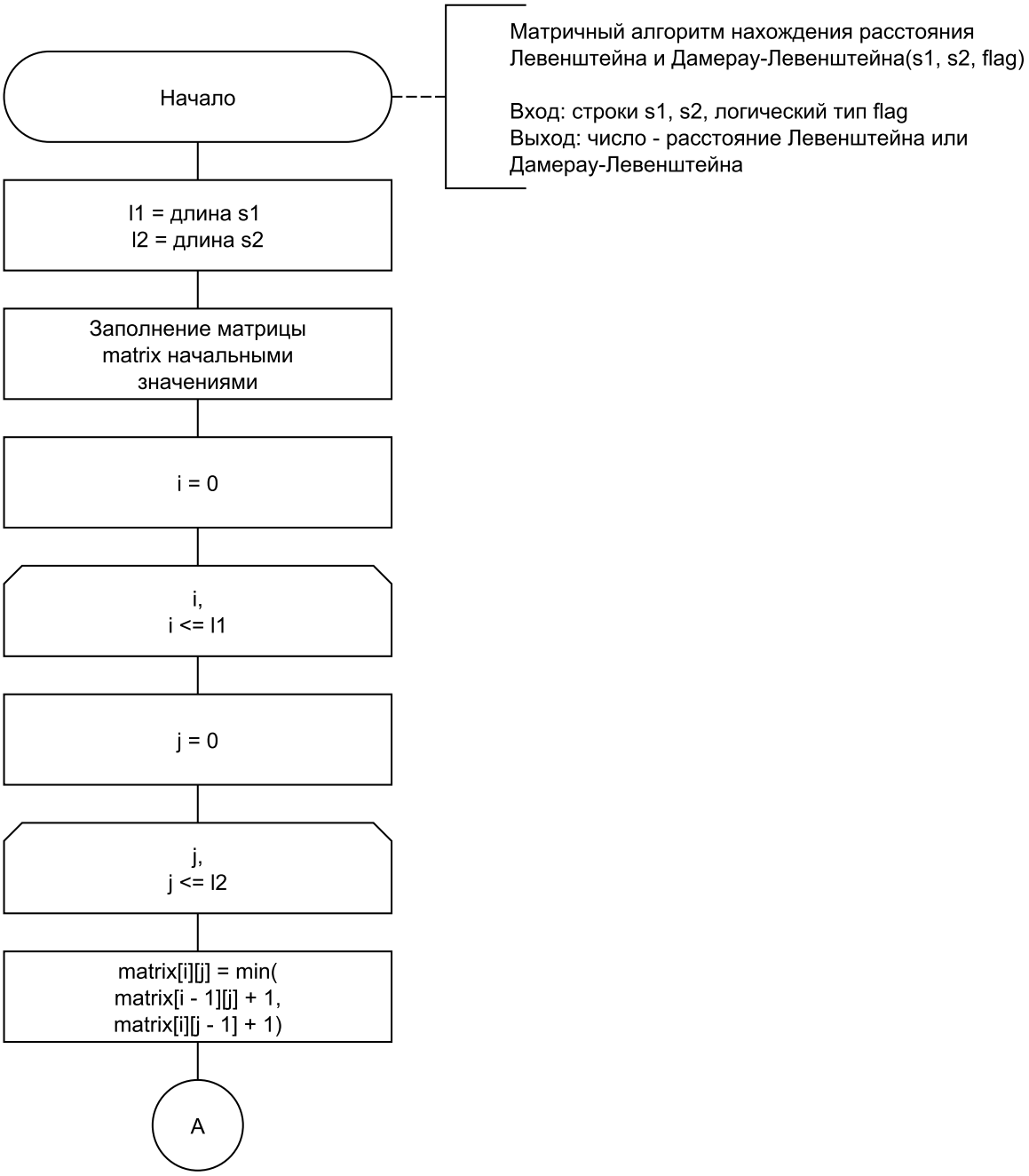

Рисунок 2.4 – Схема матричного алгоритма нахождения расстояний Левенштейна и Дамерау — Левенштейна (часть 1)

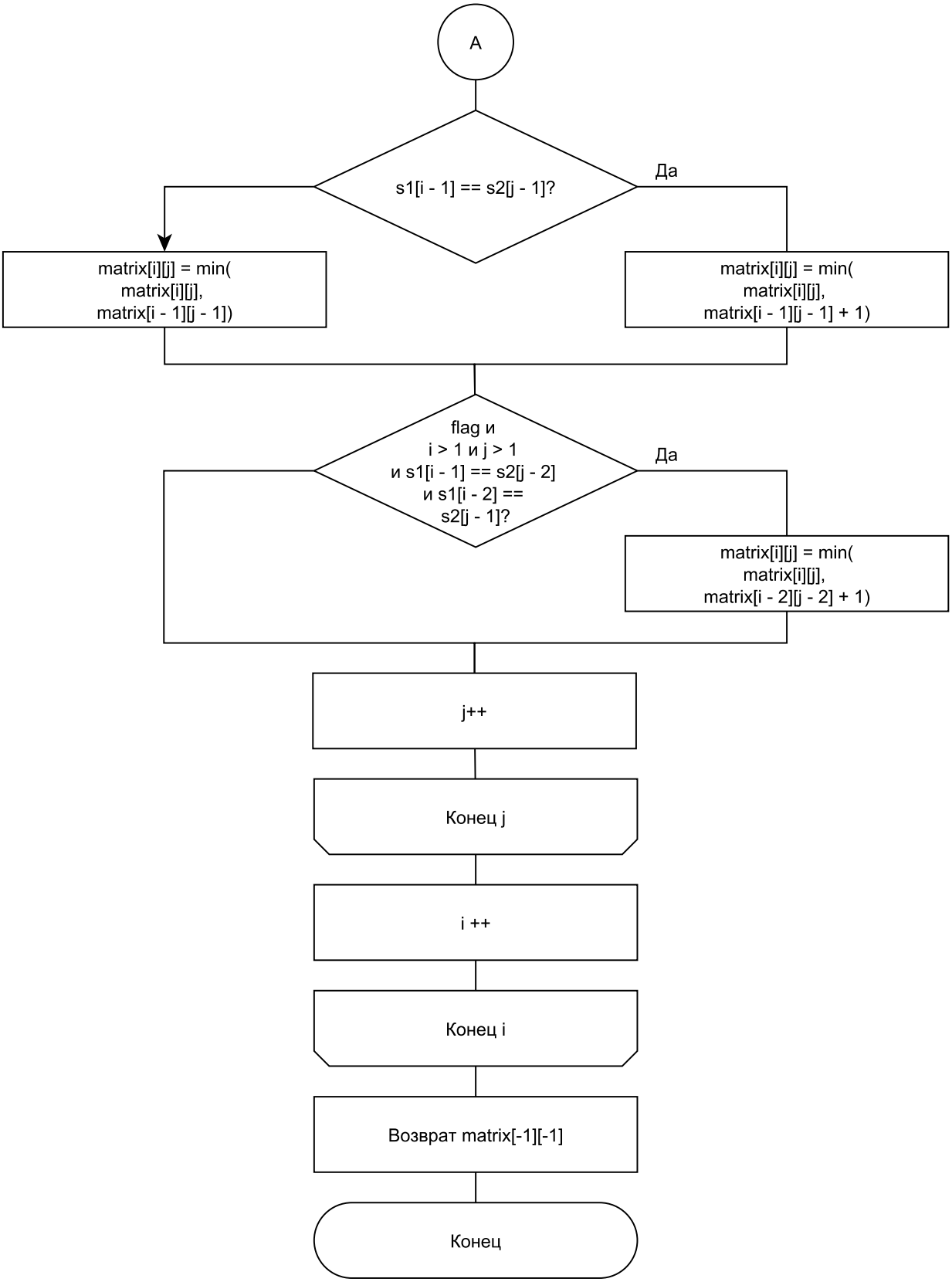

Рисунок 2.5 – Схема матричного алгоритма нахождения расстояний Левенштейна и Дамерау — Левенштейна (часть 2)

Аналогично для оптимизированного матричного алгоритма нахожде-

ния расстояний Левенштейна и Дамерау — Левенштейна. Полученная схема представлена на рисунках 2.6–2.7.

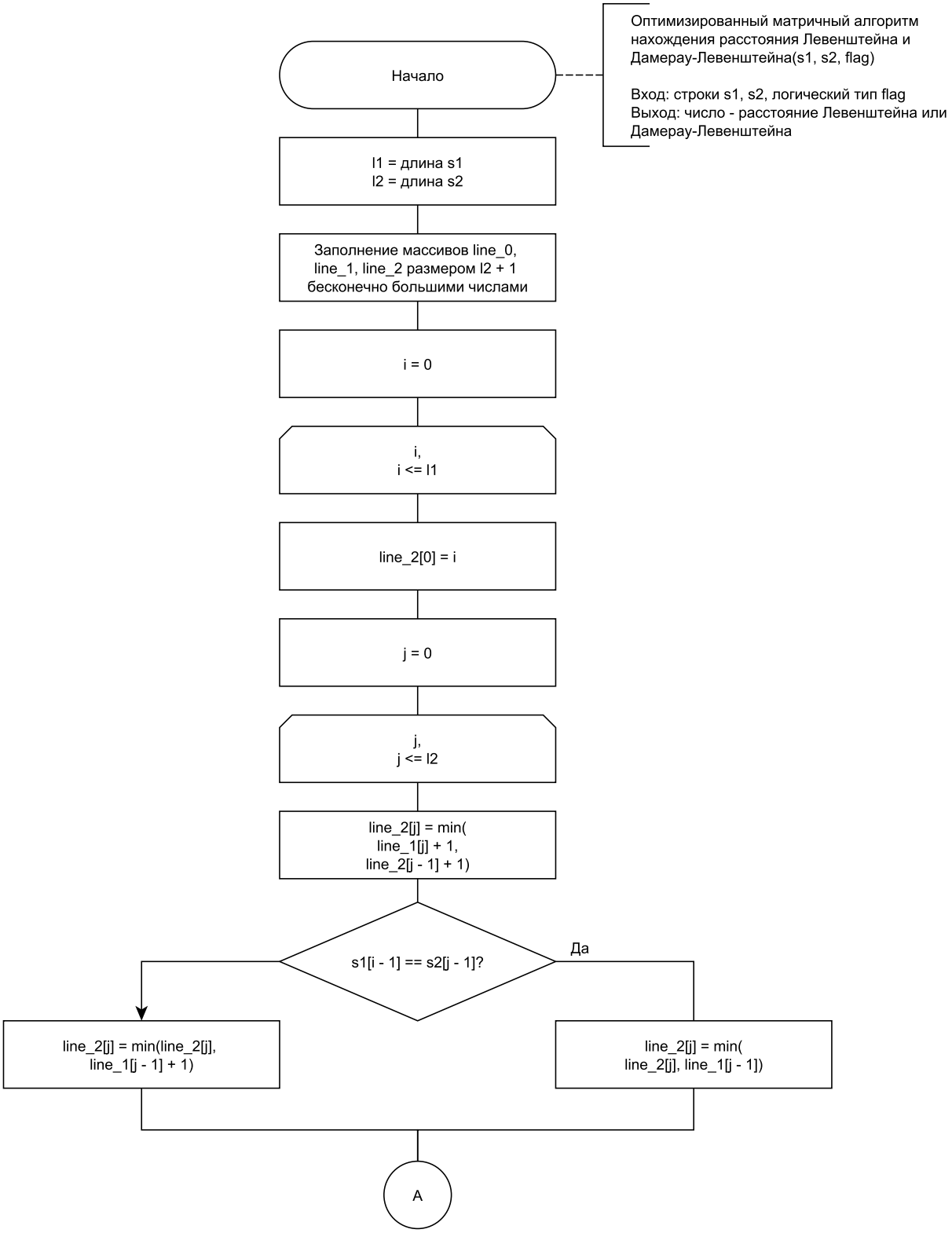

Рисунок 2.6 – Схема оптимизированного матричного алгоритма нахождения расстояний Левенштейна и Дамерау — Левенштейна (часть 1)

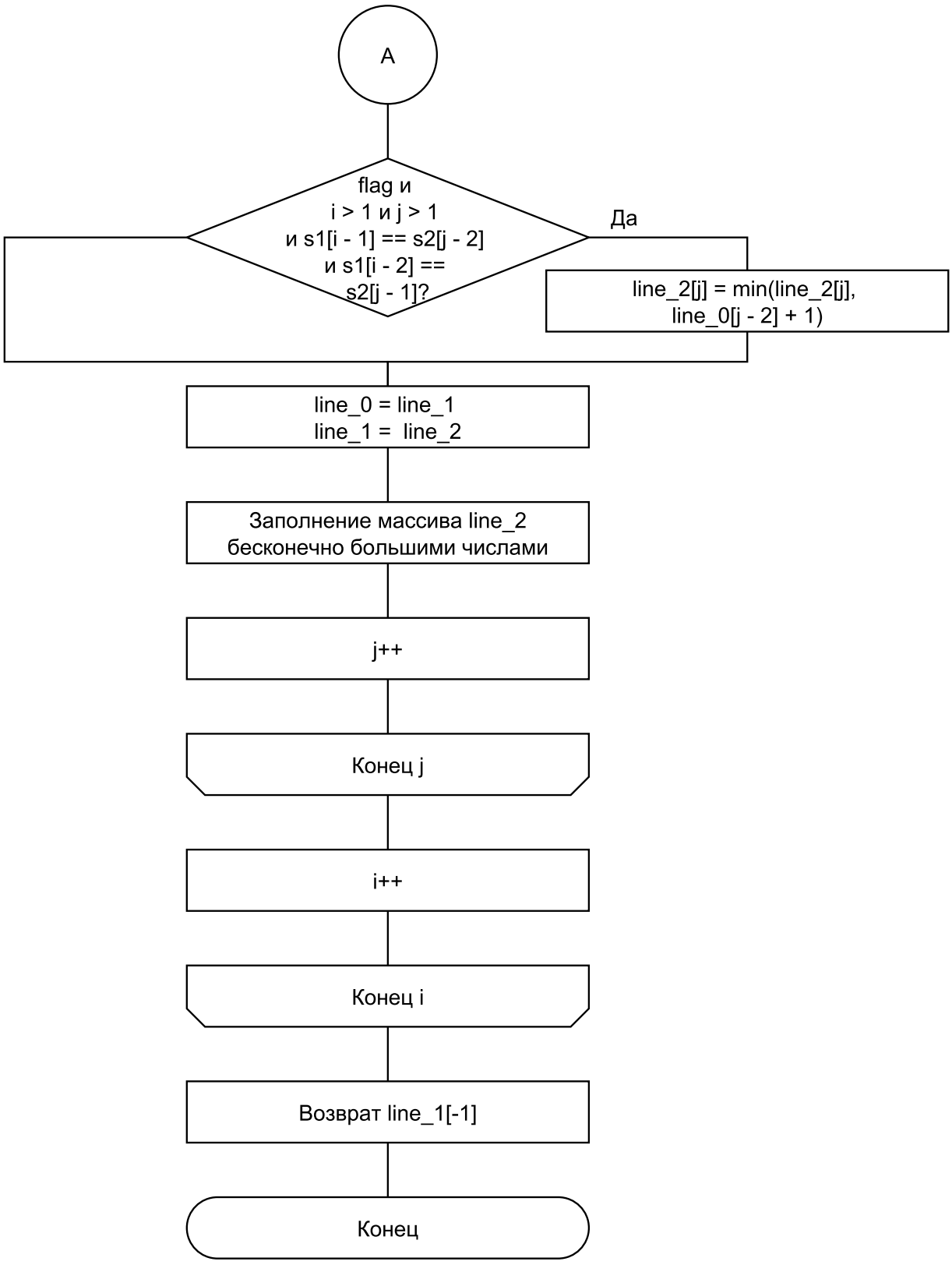

Рисунок 2.7 – Схема оптимизированного матричного алгоритма нахождения расстояний Левенштейна и Дамерау — Левенштейна (часть 2)

2.3 Классы эквивалентности тестирования

Для тестирования выделены следующие классы эквивалентности:

1. Ввод двух пустых строк;
2. Одна из строк пустая;
3. Ввод одинаковых строк;
4. Расстояния Левенштейна и Дамерау — Левенштейна равны;
5. Расстояния Левенштейна и Дамерау — Левенштейна не равны.

2.4 Использование памяти

Пусть n — длина строки S_1, m — длина строки S_2.

2.4.1 Рекурсивный алгоритм

Для каждого вызова:

— Расстояние Левенштейна:

  – для S_1, S_2: (n + m) * sizeof(char);
  – доп. переменные: 1 * sizeof(int);
  – адрес возврата.

— Затраты по памяти для нахождения расстояния Дамерау — Левенштейна равны затратам для расстояния Левенштейна.

Высота дерева вызовов: n + m.

2.4.2 Рекурсивный алгоритм с использованием кэша в виде матрицы

— Расстояние Левенштейна:

  — для матрицы: ((n + 1) * (m + 1)) * sizeof(int);

  — Для каждого вызова:

    — для S_1, S_2: (n + m) * sizeof(char);

    — для n, m: 2 * sizeof(int);

    — доп. переменные: 1 * sizeof(int);

    — ссылка на матрицу: 8 байт;

    — адрес возврата.

— Затраты по памяти для нахождения расстояния Дамерау — Левенштейна равны затратам для расстояния Левенштейна.

2.4.3 Матричный алгоритм

— Расстояние Левенштейна:

  — для матрицы: ((n + 1) * (m + 1)) * sizeof(int);

  — для S1, S2: (n + m) * sizeof(char);

  — для n, m: 2 * sizeof(int);

  — адрес возврата.

— Затраты по памяти для нахождения расстояния Дамерау — Левенштейна равны затратам для расстояния Левенштейна.

2.4.4 Оптимизированный матричный алгоритм

— Расстояние Левенштейна:

  – для S_1, S_2: (n + m) * sizeof(char);

- для n, m: 2 * sizeof(int);
- 2 строки матрицы: 2 * (n + 1) * sizeof(char);
- адрес возврата.

— Расстояние Дамерау — Левенштейна:

- для S_1, S_2: (n + m) * sizeof(char);
- для n, m: 2 * sizeof(int);
- 3 строки матрицы: 3 * (n + 1) * sizeof(char);
- адрес возврата.

Вывод

Исходя из сравнения, представленного выше, меньше всего памяти требуется для оптимизированного матричного алгоритма, так как для него необходимы только 2 строки для вычисления расстояния Левенштейна и 3 строки для вычисления расстояния Дамерау — Левенштейна.

Вывод

В данном разделе были представлено описание используемых типов данных, а также схемы алгоритмов, рассматриваемых в лабораторной работе.

3 Технологическая часть

В данном разделе будут рассмотрены средства реализации, а также представлены листинги алгоритмов определения расстояния Левенштейна и Дамерау — Левенштейна.

3.1 Средства реализации

В данной работе для реализации был выбран язык программирования *Python* [2]. В текущей лабораторной работе требуется замерить процессорное время для выполняемой программы, а также построить графики. Все эти инструменты присутствуют в выбранном языке программирования.

Время работы было замерено с помощью функции *process_ time* из библиотеки *time* [1].

3.2 Описание используемых типов данных

При реализации будут использованы следующие структуры данных:

— тип *str* для входных строк;

— тип *int* для длины строки;

— *двумерный массив ячеек типа int* — матрица для нерекурсивных алгоритмов и алгоритма с использованием кэша.

3.3 Реализация алгоритмов

В листингах 3.1–3.4 представлены реализации алгоритмов нахождения расстояний Левенштейна и Дамерау — Левенштейна. Причем расстояние Левенштейна в нерекурсивных реализациях находится с помощью той же функции, что и для Дамерау — Левенштейна, но параметр *flag* принимает значение *False*.

Листинг 3.1 – Алгоритм нахождения расстояния Дамерау–Левенштейна (рекурсивный)

```
def distance(sub_1, sub_2):
    if len(sub_1) == 0 and len(sub_2) == 0:
        return 0
    if len(sub_1) == 0 and len(sub_2) > 0:
        return len(sub_2)
    if len(sub_1) > 0 and len(sub_2) == 0:
        return len(sub_1)

    m = min(
        distance(sub_1, sub_2[:-1]) + 1,
        distance(sub_1[:-1], sub_2) + 1,
        distance(sub_1[:-1], sub_2[:-1]) + (0 if sub_1[-1] ==
           sub_2[-1] else 1)
        )

    # transpose is possible
    if len(sub_1) > 1 and len(sub_2) > 1 and sub_1[-2] == sub_2[-1]
       and sub_1[-1] == sub_2[-2]:
        return min(m, distance(sub_1[:-2], sub_2[:-2]) + 1)

    return m
```

Листинг 3.2 – Алгоритм нахождения расстояния Дамерау–Левенштейна (рекурсивный с кэшем)

```
def recursive_with_cache(s1, s2, output=True):
    def get_value(cache, i, j):
        if i == 0 and j == 0:
            cache[i][j] = 0
            return 0
        if i == 0 and j > 0:
            cache[i][j] = j
            return j
        if i > 0 and j == 0:
            cache[i][j] = i
            return i
        if cache[i][j] < float('inf'):
            return cache[i][j]

```

```
15          cache[i][j] = min(
16              get_value(cache, i, j - 1) + 1,
17              get_value(cache, i - 1, j) + 1,
18              get_value(cache, i - 1, j - 1) + (0 if s1[i - 1] ==
                    s2[j - 1] else 1)
19              )
20          if i > 1 and j > 1 and s1[i - 1] == s2[j - 2] and s1[i - 2]
                == s2[j - 1]:
21              cache[i][j] = min(cache[i][j], get_value(cache, i - 2,
                    j - 2) + 1)
22          return cache[i][j]
23
24      l1, l2 = len(s1), len(s2)
25      cache = get_matrix(l1, l2)
26
27      result = get_value(cache, l1, l2)
28      if output:
29          print_matrix(s1, s2, cache)
30
31      return result
```

Листинг 3.3 – Алгоритм нахождения расстояния Дамерау–Левенштейна (матричный)

```
1  def full_matrix(s1, s2, flag, output=True):
2      l1, l2 = len(s1), len(s2)
3      matrix = get_matrix(l1, l2)
4
5      for i in range(1, l1 + 1):
6          for j in range(1, l2 + 1):
7              matrix[i][j] = min(
8                  matrix[i - 1][j] + 1,
9                  matrix[i][j - 1] + 1,
10                 matrix[i - 1][j - 1] + (0 if s1[i - 1] == s2[j - 1]
                       else 1)
11                 )
12             if flag and i > 1 and j > 1 and s1[i - 1] == s2[j - 2]
                   and s1[i - 2] == s2[j - 1]:
13                 matrix[i][j] = min(matrix[i][j], matrix[i - 2][j -
                       2] + 1)
14
15     if output:
```

```
16      print_matrix(s1, s2, matrix)
17
18      return matrix[-1][-1]
```

Листинг 3.4 – Алгоритм нахождения расстояния Дамерау–Левенштейна (оптимизированный матричный)

```
1  def optimised(s1, s2, flag):
2      l1, l2 = len(s1), len(s2)
3      # line_2 - current
4      line_0, line_1, line_2 = [float('inf')] * (l2 + 1), [i for i in
           range(l2 + 1)], [float('inf')] * (l2 + 1)
5
6      for i in range(1, l1 + 1):
7          line_2[0] = i
8          for j in range(1, l2 + 1):
9              line_2[j] = min(
10                 line_1[j] + 1,
11                 line_2[j - 1] + 1,
12                 line_1[j - 1] + (0 if s1[i - 1] == s2[j - 1] else 1)
13                 )
14             if flag and i > 1 and j > 1 and s1[i - 1] == s2[j - 2]
                   and s1[i - 2] == s2[j - 1]:
15                 line_2[j] = min(line_2[j], line_0[j - 2] + 1)
16
17         line_0, line_1, line_2 = line_1, line_2, [float('inf')] *
               (l2 + 1)
18
19     return line_1[-1]
```

3.4 Функциональные тесты

В таблице 3.1 приведены тесты для функций, реализующих алгоритмы нахождения расстояния Левенштейна и Дамерау — Левенштейна. Тесты *для всех алгоритмов* 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"	слово	5	5
3	проверка	"пустая строка"	8	8
4	ремонт	емонт	1	1
5	гигиена	иена	3	3
6	слон	салон	1	1
7	спасибо	пожалуйста	9	9
8	что	кто	1	1
9	ты	тыква	3	3
10	есть	кушать	4	4
11	abba	baab	3	2
12	abcba	bacab	4	2

Вывод

Были представлены всех алгоритмов нахождения расстояний Левенштейна и Дамерау — Левенштейна, которые были описаны в предыдущем разделе. Также в данном разделе была приведена информации о выбранных средствах для разработки алгоритмов.

4 Исследовательская часть

В данном разделе будут приведены примеры работы программы, а также проведен сравнительный анализ алгоритмов при различных ситуациях на основе полученных данных.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование представлены далее:

— операционная система: Windows 11, x64 [3];

— оперативная память: 8 Гб;

— процессор: AMD Ryzen 5 5500U с видеокартой Radeon Graphics 2.10 ГГц.

Во время замеров времени ноутбук был нагружен только встроенными приложениями окружения.

4.2 Демонстрация работы программы

На рисунке 4.1 представлен результат работы программы.

```
Расстояние Дамерау-Левенштейна:
    1) Нерекурсивно
    2) Рекурсивно
    3) Рекурсивно с кэшем
Расстояние Левенштейна:
    4) Нерекурсивно

    5) Построить график
    0) Выйти

Введите номер: 1
    1) Используя всю матрицу
    2) Используя 3 последних строки
Введите номер: 1
Введите первую строку: абвгд
Введите вторую строку: баоуд

Полученная матрица:
0  0  б  а  о  у  д
0  0  1  2  3  4  5
а  1  1  1  2  3  4
б  2  1  1  2  3  4
в  3  2  2  2  3  4
г  4  3  3  3  3  4
д  5  4  4  4  4  3
Полученное расстояние: 3
```

Рисунок 4.1 – Пример работы программы

4.3 Время выполнения алгоритмов

Как было сказано выше, используется функция замера процессорного времени *process_time* из библиотеки *time* на *Python*. Функция возвращает пользовательское процессорное время типа *float*.

Использовать функцию приходится дважды, затем из конечного времени нужно вычесть начальное, чтобы получить результат.

Замеры проводились для длины слова от 10 до 410 с шагом 50 по 100 раз на различных входных данных.

Результаты замеров приведены в таблице 4.1 (время в мс). Используются следующие обозначения:

— Д.-Л.(матр.) — матричный алгоритм нахождения расстояния Дамерау — Левенштейна;

— Л.(матр.) — матричный алгоритм нахождения расстояния Левенштейна;

— Д.-Л.(опт.) — оптимизированный матричный алгоритм нахождения расстояния Дамерау — Левенштейна;

— Л.(опт.) — оптимизированный матричный алгоритм нахождения расстояния Левенштейна;

— Д.-Л.(кэш) — рекурсивный алгоритм нахождения расстояния Дамерау — Левенштейна с использованием кэша в виде матрицы.

Таблица 4.1 – Результаты замеров времени

Длина	Д.-Л.(матр.), мс	Л.(матр.), мс	Д.-Л.(опт.), мс	Л.(опт.), мс	Д.-Л.(кэш), мс
10	0.0588	0.0550	0.0503	0.0397	0.1522
60	2.0313	1.7813	1.7188	1.2969	5.0781
110	6.3750	5.9218	6.0313	4.1250	18.8125
160	13.7500	10.9375	13.5938	7.6563	41.2500
210	24.3750	18.7500	22.6563	16.7188	67.9688
260	36.7188	25.9375	32.0313	24.2188	104.3750
310	56.5625	40.6250	47.1875	35.6250	150.6250
360	76.7188	61.0938	61.4063	49.5313	211.8750
410	100.3125	78.7500	83.9063	60.3125	273.5938

На рисунке 4.2 приведена визуализация результатов замеров.

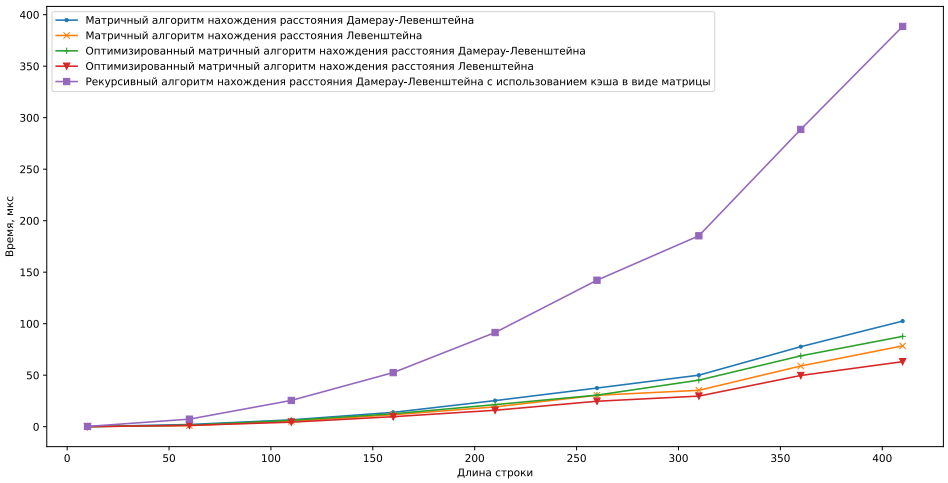

Рисунок 4.2 – Визуализация результатов замеров

4.4 Вывод

Исходя из оценки памяти, итеративные алгоритмы проигрывают рекурсивным, потому что максимальный размер памяти в них растет, как произведение длин строк, а в рекурсивных - как сумма длин строк.

В результате эксперимента было получено, что наибольшее время работы программы получается при использовании рекурсивных алгоритмов. Исходя из замеров по времени, реализация оптимизированного матричного алгоритма нахождения расстояния Левенштейна при больших длинах строк работает быстрее остальных реализаций.

Заключение

В результате исследования было определено, что время работы реализаций алгоритмов Левенштейна и Дамерау — Левенштейна растет в геометрической прогрессии при увеличении длин строк. Но лучшие показатели по времени у матричной реализации алгоритма Левенштейна и его оптимизированной по памяти реализации. Также лучшие показатели по памяти у оптимизированной реализации нахождения расстояния Левенштейна. Худшие показатели по времени работы и по памяти у рекурсивной реализации вычисления расстояния Дамерау — Левенштейна.

Цель, которая была поставлена в начале лабораторной работы была достигнута: исследованы алгоритмы нахождения расстояний Левенштейна и Дамерау — Левенштейна. В ходе выполнения были решены все задачи:

— описаны расстояния Левенштейна и Дамерау — Левенштейна;

— реализованы алгоритмы нахождения расстояний Левенштейна и Дамерау — Левенштейна;

— проведено тестирование по методу черного ящика для реализаций алгоритмов нахождения расстояний Левенштейна и Дамерау — Левенштейна;

— проведен сравнительный анализ по времени рекурсивной и матричной реализации алгоритма нахождения расстояния Левенштейна;

— проведен сравнительный анализ по времени матричной и с кешем реализации алгоритма нахождения расстояния Левенштейна;

— проведен сравнительный анализ по времени алгоритмов нахождения расстояния Левенштейна и Дамерау — Левенштейна;

— подготовлен отчет о лабораторной работе.

Список используемых источников

1. time — Time access and conversions [Электронный ресурс]. — URL: https://docs.python.org/3/library/time.html#functions (дата обр. 20.09.2023).
2. Welcome to Python [Электронный ресурс]. — URL: https://www.python.org (дата обр. 20.09.2023).
3. Windows 11 Home. — URL: https://www.microsoft.com/en-us/windows/get-windows-11?activetab=pivot%3aoverviewtab&r=1.
4. Погорелов, Д. А. Сравнительный анализ алгоритмов редакционного расстояния Левенштейна и Дамерау-Левенштейна / Д. А. Погорелов, А. М. Таразанов. - Синергия Наук. – 2019. — URL: https://elibrary.ru/item.asp?id=36907767 (дата обр. 20.09.2023).