Содержание

Введение

В данной лабораторной работе будет рассмотрено расстояние Левенштейна. Данное расстояние показывает минимальное количество операций (вставка, удаление, замены), которое необходимо для перевода одной строки в другую. Это расстояние помогает определить схожесть двух строк.

Впервые задачу поставил в 1965 году советский математик Владимир Левенштейн при изучении последовательностей 0 — 1, впоследствии более общую задачу для произвольного алфавита связали с его именем.

Расстояние Левенштейна применяется в теории информации и компьютерной лингвистике для решения следующих задач:

— исправление ошибок в слове(в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи);

— сравнение текстовых файлов утилитой diff;

— для сравнения геномов, хромосом и белков в биоинформатике.

Метод динамического программирования [1] был предложен и обоснован Р. Беллманом в начале 1960-х годов. Первоначально метод создавался в целях существенного сокращения перебора для решения целого ряда задач экономического характера, формулируемых в терминах задач целочисленного программирования. Однако Р. Беллман и Р. Дрейфус показали, что он применим к достаточно широкому кругу задач, в том числе к задачам поиска расстояния Левенштейна и Дамерау-Левенштейна.

Целью данной лабораторной работы является описание и исследование алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.

Для поставленной цели необходимо выполнить следующие задачи.

1) Описать алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна.

2) Создать программное обеспечение, реализующее следующие алгоритмы:

   — нерекурсивный алгоритм поиска расстояния Левенштейна;

   — нерекурсивный алгоритм поиска расстояния Дамерау-Левенштейна;

   — рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна;

   — рекурсивный с кэшированием алгоритм поиска расстояния Дамерау-Левенштейна.

3) Выбрать инструменты для замера процессорного времени выполнения реализаций алгоритмов.

4) Провести анализ затрат реализаций алгоритмов по времени и по памяти, определить влияющие на них характеристики.

1 Аналитическая часть

1.1 Расстояние Левенштейна

Расстояние Левенштейна [2] (редакционное расстояние, дистанция редактирования) — метрика, измеряющая разность между двумя последовательностями символов. Расстояние Левенштейна — это минимальное количество редакторских операций вставки (I, от англ. insert), замены (R, от англ. replace) и удаления (D, от англ. delete), необходимых для преобразования одной строки в другую. Стоимости операций могут зависеть от вида операций:

1) $w(a, b)$ — цена замены символа a на b;

2) $w(\lambda, b)$ — цена вставки символа b;

3) $w(a, \lambda)$ — цена удаления символа a.

Считаем, что стоимость каждой операции равной 1:

— $w(a, b) = 1$, $a \neq b$, в противном случае замена не происходит;

— $w(\lambda, b) = 1$;

— $w(a, \lambda) = 1$.

Введем понятие совпадения символов — M (от англ. match). Его стоимость будет равна 0, то есть $w(a, a) = 0$.

Введем в рассмотрение функцию $D(i, j)$, значением которой является редакционное расстояние между подстроками $S_1[1...i]$ и $S_2[1...j]$.

Расстояние Левенштейна между двумя строками S_1 и S_2 длиной M

и N соответственно рассчитывается по рекуррентной формуле:

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

где сравнение символов строк S_1 и S_2 рассчитывается как:

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

1.1.1 Нерекурсивный алгоритм нахождения расстояния Левенштейна

Рекурсивная реализация алгоритма Левенштейна малоэффективна по времени при больших M и N, так как множество промежуточных значений Для оптимизации можно использовать итерационную реализацию заполнения матрицы промежуточными значениями $D(i,j)$.

В качестве структуры данных для хранения промежуточных значений можно использовать матрицу, имеющую размеры:

$$(N+1) \times (M+1) \tag{1.3}$$

Значения в ячейке $[i,j]$ равно значению $D(S1[1...i], S2[1...j])$. Первый элемент матрицы заполнен нулем. Всю таблицу заполнять в соответствии с формулой (1.1).

Однако матричный алгоритм является малоэффективным по памяти по сравнению с рекурсивным при больших M и N, т.к. множество промежуточных значений $D(i,j)$ хранится в памяти после их использования. Для оптимизации по памяти рекурсивного алгоритма

нахождения расстояния Левенштейна можно использовать кеш, т.е. пару строк, содержащую значения $D(i,j)$, вычисленные в предыдущей итерации, алгоритма и значения $D(i,j)$, вычисляемые в текущей итерации.

1.2 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна, названное в честь ученых Фредерика Дамерау и Владимир 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 трем базовым операциям добавляется операция транспозиции T (от англ. transposition).

Расстояние Дамерау-Левенштейна может быть вычислено по рекуррентной формуле:

$$D(i,j) = \begin{cases} 0, & \text{i} = 0,\ \text{j} = 0, \\ i, & \text{j} = 0,\ \text{i} > 0, \\ j, & \text{i} = 0,\ \text{j} > 0, \\ min \begin{cases} D(i,j-1)+1, \\ D(i-1,j)+1, \\ D(i-1,j-1)+m(S_1[i],S_2[j]), \\ D(i-2,j-2)+1, \end{cases} & \text{если i} > 1,\ \text{j} > 1, \\ & S_1[i] = S_2[j-1], \\ & S_1[i-1] = S_2[j], \\ min \begin{cases} D(i,j-1)+1, \\ D(i-1,j)+1, \\ D(i-1,j-1)+m(S_1[i],S_2[j]), \end{cases} & \text{иначе.} \end{cases} \tag{1.4}$$

1.2.1 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна

Рекурсивный алгоритм реализует формулу (1.4), функция D составлена таким образом, что верно следующее.

1) Для передачи из пустой строки в пустую требуется ноль операций.

2) Для перевода из пустой строки в строку a требуется $|a|$ операций.

3) Для перевода из строки a в пустую строку требуется $|a|$ операций.

4) Для перевода из строки a в строку b требуется выполнить последовательно некоторое количество операций удаления, вставки, замены, транспозиции в некоторой последовательности. Последовательность поведения любых двух операций можно поменять, порядок поведения операций не имеет никакого значения. Если полагать, что *a', b'* – строки a и b без последнего символа соответственно, а *a", b"* – строки a и b без двух последних символов, то цена преобразования из строки a в b выражается из элементов, представленных ниже:

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

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

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

   — сумма цены преобразования из a'' в b'' и операции перестановки, предполагая, что длины a'' и b'' больше 1 и последние два символа a'', поменянные местами, совпадут с двумя последними символами b'';

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

Минимальной стоимостью преобразования будет минимальное значение приведенных вариантов.

1.2.2 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна с кэшированием

Рекурсивная реализация алгоритма Дамерау-Левенштейна малоэффективна по времени при больших M и N по причине проблемы повторных вычислений значений расстояний между подстроками. Для оптимизации алгоритма нахождения расстояния Левенштейна можно использовать матрицу в целях хранения соответствующих промежуточных значений. В таком случае алгоритм представляет собой рекурсивное заполнение матрицы $A_{|a|,|b|}$ промежуточными значениями $D(i,j)$, такое хранение промежуточных данных можно назвать кэшем для рекурсивного алгоритма.

1.2.3 Нерекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна

Рекурсивная реализация алгоритма Левенштейна с кешированием малоэффективна по времени при больших M и N. Для оптимизации можно использовать итерационную реализацию заполнения матрицы промежуточными значениями $D(i,j)$.

В качестве структуры данных для хранения промежуточных значений можно использовать *матрицу*, имеющую размеры:

$$(N+1) \times (M+1), \tag{1.5}$$

Значение в ячейке $[i,j]$ равно значению $D(S1[1...i], S2[1...j])$. Первый

элемент заполнен нулем. Всю таблицу заполняем в соответствии с формулой (1.4).

Вывод

В данном разделе были рассмотрены алгоритмы динамического программирования — алгоритмы нахождения расстояний Левенштейна и Дамерау-Левенштейна, формулы которых задаются рекуррентно, а следовательно, данные алгоритмы могут быть реализованы рекурсивно и итеративно. На вход алгоритмам поступают две строки, которые могут содержать как русские, так и английские буквы, также будет предусмотрен ввод пустых строк.

2 Конструкторская часть

В данном разделе будут приведены схемы алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна, приведены описание используемых типов данных, оценки памяти, а также описана структура программного обеспечения.

2.1 Требования к программному обеспечению

К программе предъявлен ряд функциональных требований: входные данные — две строки, выходные данные — результат работы всех алгоритмов поиска расстояний, целое число.

К программе предъявлен ряд требований:

— наличие интерфейса для выбора действий;

— должна обрабатывать строки;

— возможность обработки строк, включающих буквы как на латинице, так и на кириллице;

— наличие функциональности замера процессорного времени работы реализаций алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.

2.2 Разработка алгоритмов

На вход алгоритмов подаются строки S_1 и S_2.

На рисунке 2.1 представлена схема алгоритма поиска расстояния Левенштейна. На рисунках 2.2 – 2.5 представлены схемы алгоритмов поиска Дамерау-Левенштейна.

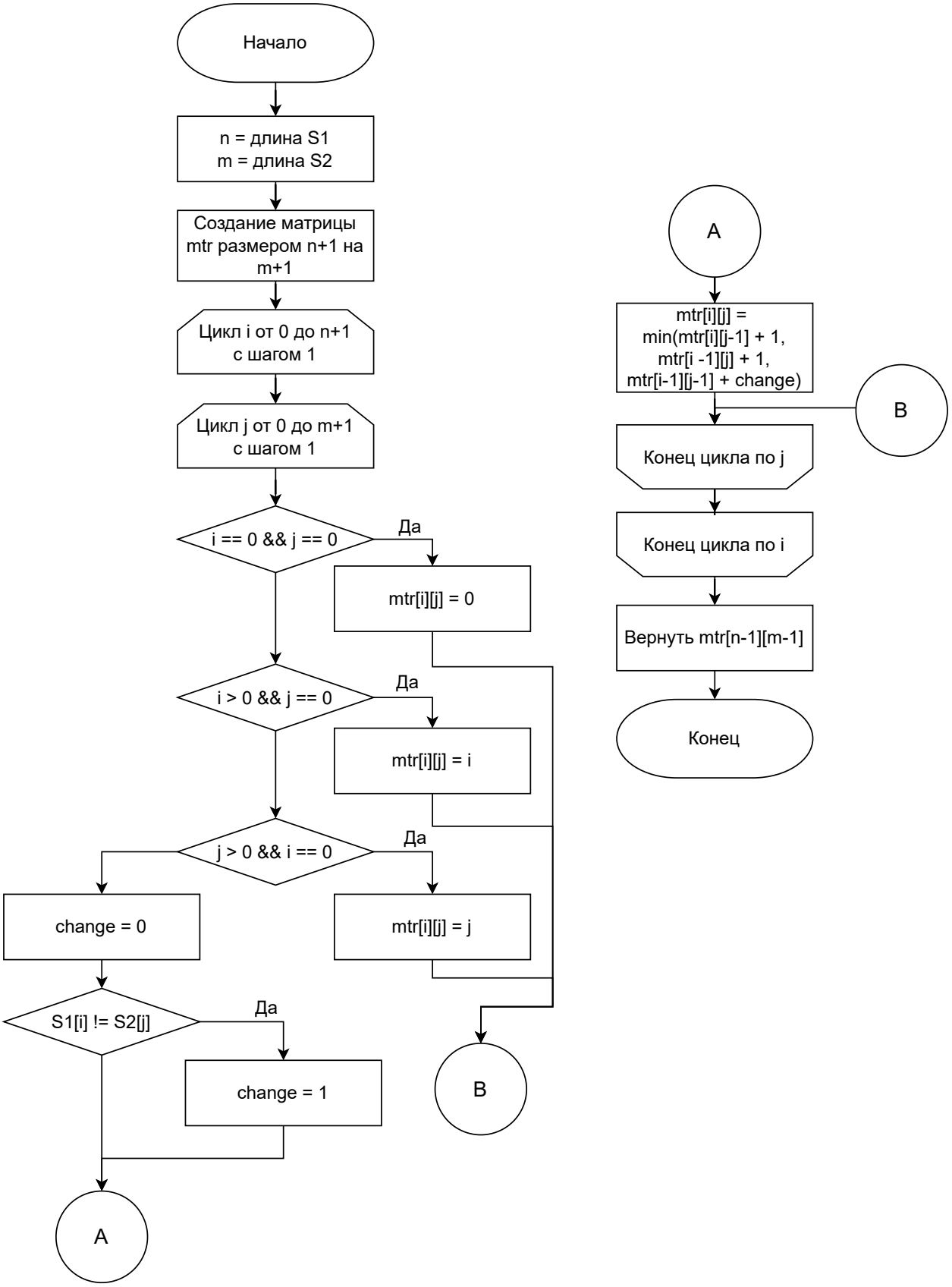

Рисунок 2.1 – Схема нерекурсивного алгоритма нахождения расстояния Левенштейна

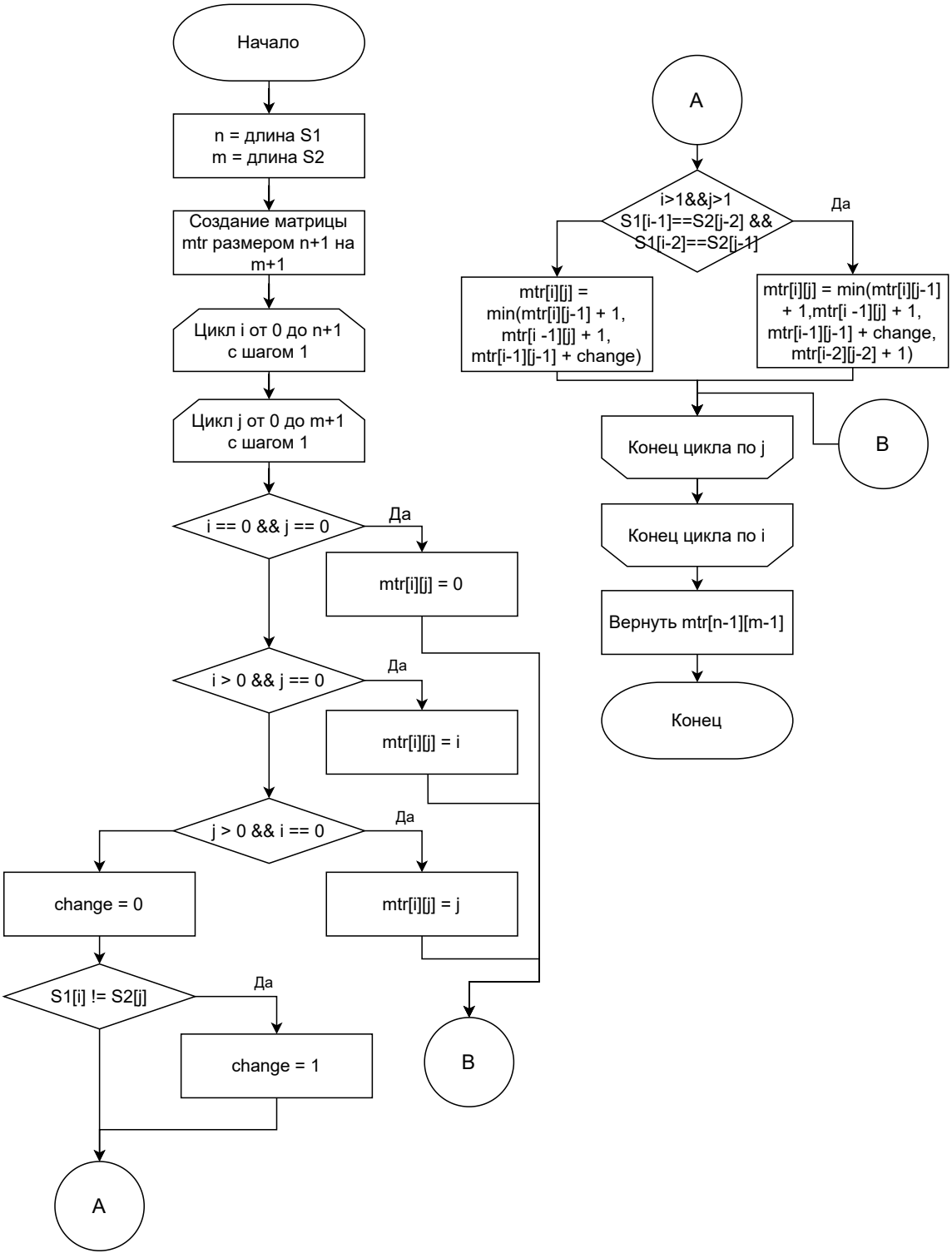

Рисунок 2.2 – Схема нерекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

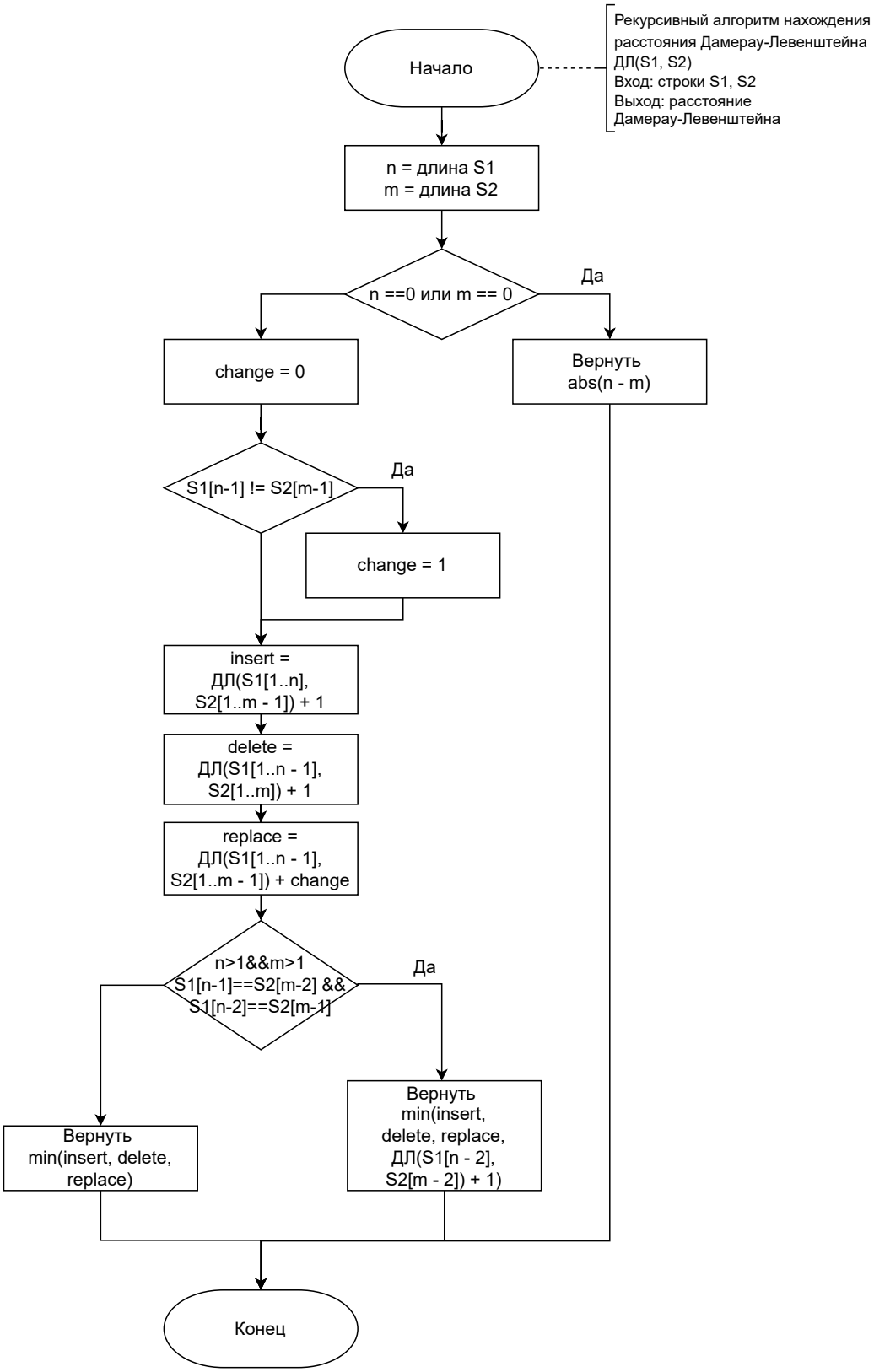

Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

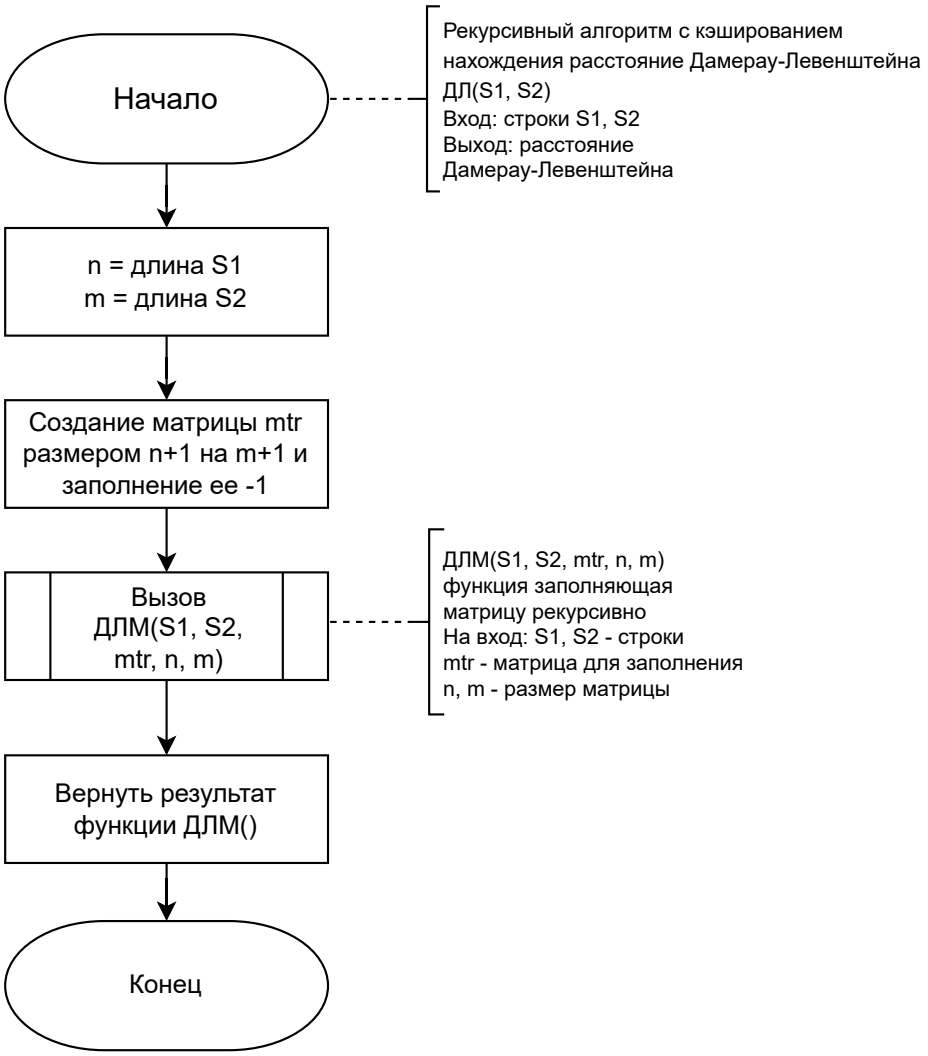

Рисунок 2.4 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с кэшированием

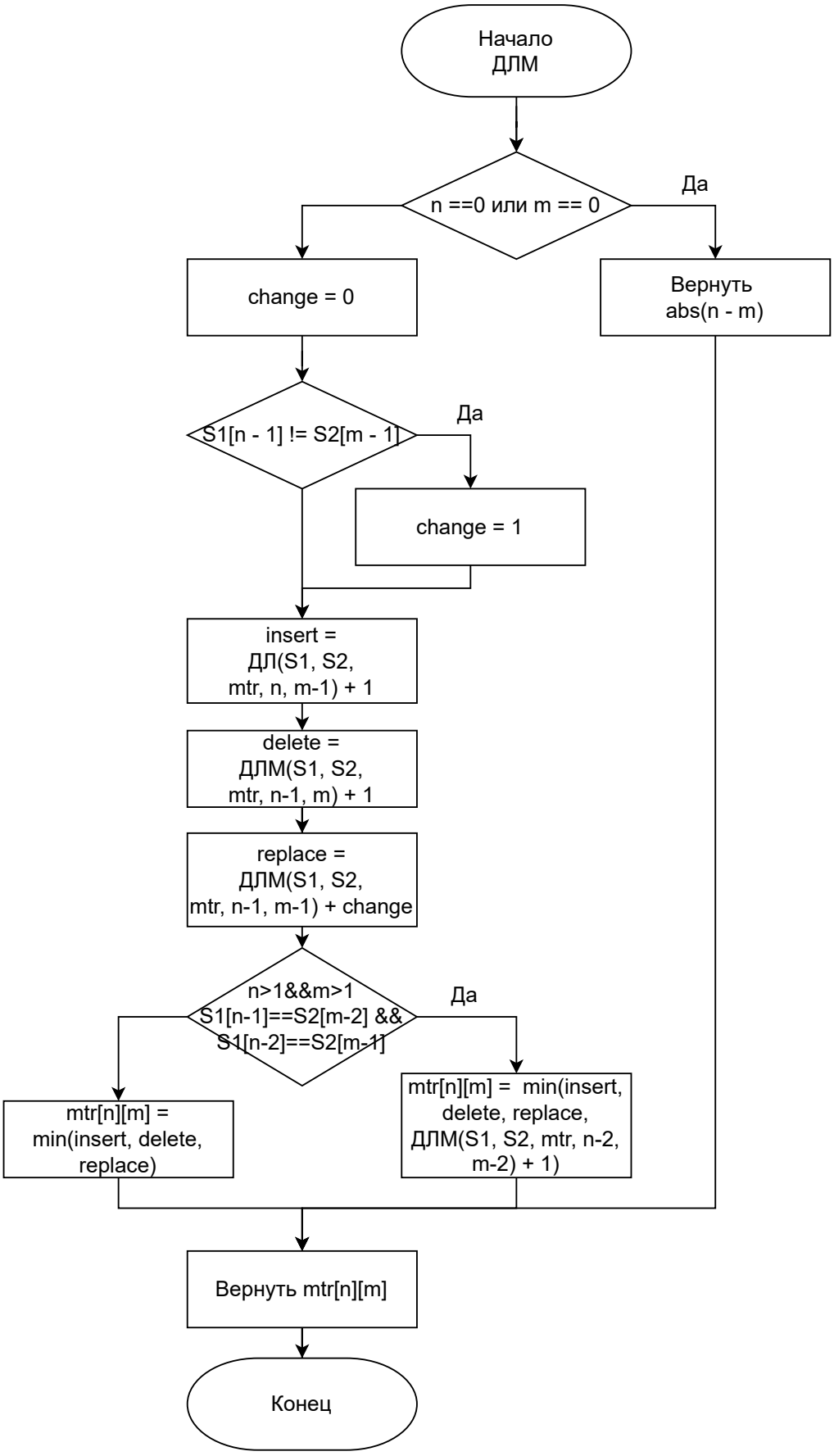

Рисунок 2.5 – Схема алгоритма рекурсивного заполнения матрицы путем поиска расстояния Дамерау-Левенштейна

Вывод

В данном разделе на основе теоретических данных были построены схемы требуемых алгоритмов, выбраны используемые типы данных.

3 Технологическая часть

В данном разделе будут приведены требования к программному обеспечению, средства реализации, листинг кода и функциональные тесты.

3.1 Средства реализации

Для реализации данной лабораторной работы был выбран язык *Python* [3]. Данный выбор обусловлен наличием у языка функции $processor_time()$ измерения процессорного времени и типа данных, позволяющего хранить как кириллические символы, так и латинские — *string*, что позволит удовлетворить третьему требованию из п.2.1 соответствуют выдвинутым техническим требованиям. Также в языке присутствует метод $sys.getsizeof()$, позволяющий определить размер памяти, занимаемый объектом в программе.

3.2 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие структуры данных:

— строка (*string*) — массив символов;

— длина строки — целое число типа *int*;

— матрица — $list(list())$

3.3 Реализация алгоритмов

В листингах 3.1 – 3.4 приведены реализации алгоритмов поиска расстояний Левенштейна (только нерекурсивный алгоритм) и Дамерау-Левенштейна (нерекурсивный, рекурсивный и рекурсивный с кешированием).

В листингах 3.5 – 3.6 приведены реализации алгоритмов вывода промежуточной информации на экран.

В листингах 3.7 – 3.8 приведены реализации алгоритмов подсчета процессорного времени, затраченного на выполнение функции, и генерации случайной строки.

В листингах 3.9 – 3.13 приведены алгоритмы замера функций, создания csv-таблиц и графиков.

Листинг 3.1 – Функция нахождения расстояния Левенштейна с использованием матрицы

```
def lev(s1, s2, print_flag=False):
    n = len(s1) + 1
    m = len(s2) + 1
    mat = [[0] * m for _ in range(n)]

    for i in range(n):
        for j in range(m):
            if i == 0 and j == 0:
                mat[i][j] = 0
            elif i > 0 and j == 0:
                mat[i][j] = i
            elif i == 0 and j > 0:
                mat[i][j] = j
            else:
                mat[i][j] = min(mat[i][j - 1] + 1, mat[i -
                    1][j] + 1, mat[i - 1][j - 1] + int(s1[i - 1]
                    != s2[j - 1]))

    if print_flag:
        print_mat(mat, s1, s2)

    return mat[n - 1][m - 1]
```

Листинг 3.2 – Функция нахождения расстояния Дамерау-Левенштейна с использованием матрицы

```
1  def dam_lev(s1, s2, print_flag=False):
2      n = len(s1) + 1
3      m = len(s2) + 1
4      mat = [[0] * m for _ in range(n)]
5
6      for i in range(n):
7          for j in range(m):
8              if i == 0 and j == 0:
9                  mat[i][j] = 0
10             elif i > 0 and j == 0:
11                 mat[i][j] = i
12             elif i == 0 and j > 0:
13                 mat[i][j] = j
14             else:
15                 mat[i][j] = min(mat[i][j - 1] + 1, mat[i -
                       1][j] + 1, mat[i - 1][j - 1] + int(s1[i - 1]
                       != s2[j - 1]))
16                 if i > 1 and j > 1 and s1[i - 1] == s2[j - 2]
                       and s1[i - 2] == s2[j - 1]:
17                     mat[i][j] = min(mat[i][j], mat[i - 2][j -
                           2] + 1)
18
19     if print_flag:
20         print_mat(mat, s1, s2)
21
22     return mat[n - 1][m - 1]
```

Листинг 3.3 – Функция нахождения расстояния Дамерау-Левенштейна рекурсивно

```
1  def dam_lev_rec_t(s1, s2, i, j):
2      if i == 0:
3          return j
4      if j == 0:
5          return i
6
7      res = min(dam_lev_rec_t(s1, s2, i, j - 1) + 1,
           dam_lev_rec_t(s1, s2, i - 1, j) + 1,
8                  dam_lev_rec_t(s1, s2, i - 1, j - 1) + int(s1[i -
                       1] != s2[j - 1]))
9
10     if i > 1 and j > 1 and s1[i - 1] == s2[j - 2] and s1[i - 2]
           == s2[j - 1]:
11         res = min(res, dam_lev_rec_t(s1, s2, i - 2, j - 2) + 1)
12
13     return res
14
15
16 def dam_lev_rec(s1, s2):
17     return dam_lev_rec_t(s1, s2, len(s1), len(s2))
```

Листинг 3.4 – Функция нахождения расстояния Дамерау-Левенштейна рекурсивно с кешированием

```
def dam_lev_rec_hash_t(s1, s2, i, j, mat):
    if i == 0:
        mat[i][j] = j
        return j
    if j == 0:
        mat[i][j] = i
        return i

    res = min(
        (mat[i][j - 1] if mat[i][j - 1] != -1 else
           dam_lev_rec_hash_t(s1, s2, i, j - 1, mat)) + 1,
        (mat[i - 1][j] if (mat[i - 1][j] != -1) else
           dam_lev_rec_hash_t(s1, s2, i - 1, j, mat)) + 1,
        (mat[i - 1][j - 1] if (mat[i - 1][j - 1] != -1) else
           dam_lev_rec_hash_t(s1, s2, i - 1, j - 1, mat)) + int(
            s1[i - 1] != s2[j - 1])
    )

    if i > 1 and j > 1 and s1[i - 1] == s2[j - 2] and s1[i - 2]
       == s2[j - 1]:
        res = min(res, (
            mat[i - 2][j - 2] if (mat[i - 2][j - 2] != -1) else
               dam_lev_rec_hash_t(s1, s2, i - 2, j - 2, mat)) +
               1)

    mat[i][j] = res
    return res

def dam_lev_rec_hash(s1, s2, print_flag=False):
    n = len(s1)
    m = len(s2)
    mat = [[0] * (m + 1) for _ in range(n + 1)]

    res = dam_lev_rec_hash_t(s1, s2, n, m, mat)

    if print_flag:
        print_mat(mat, s1, s2)

    return res
```

Листинг 3.5 – Функции вывода матрицы для алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна

```
def print_mat(mat, s2, s1):
    print("\n\t-\t", end='')
    for i in s1:
        print(f"{i}\t", end='')
    print()
    for i in range(len(s2) + 1):
        if i == 0:
            print("-\t", end='')
        else:
            print(f"{s2[i - 1]}\t", end='')
        for j in range(len(s1) + 1):
            print(f"{mat[i][j]}\t", end='')
        print()
```

Листинг 3.6 – Функция вывода меню на экран

```
def print_menu():
    print(
        "Меню:\n"
        "0) Выйти\n"
        "1) Посчитать матричным алгоритмом Левенштейна с печатанием таблицы\n"
        "2) Посчитать матричным алгоритмом Дамерау-Левенштейна с печатанием таблицы\n"
        "3) Посчитать рексурсивным алгоритмом Дамерау-Левенштейна\n"
        "4) Посчитать рекурсивным с хэшем алгоритмом Дамерау-Левенштейна с печатанием таблицы\n"
        "5) Сравнить алгоритмы"
    )
```

Листинг 3.7 – Функция подсчета процессорного времени

```
1  TIMES = 100
2  def get_process_time(func, size, times=TIMES):
3      time_res = 0
4
5      for _ in range(times):
6          str1 = get_random_string(size)
7          str2 = get_random_string(size)
8
9          time_start = process_time()
10         func(str1, str2)
11         time_end = process_time()
12
13         time_res += (time_end - time_start)
14
15     return time_res / times
```

Листинг 3.8 – Функция генерации случайных строк

```
1  def get_random_string(size):
2      buf = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ" \
3            "abcdefghijklmnopqrstuvwxyz"
4
5      return "".join(choice(buf) for _ in range(size))
```

Листинг 3.9 – Функция получения графика затраченной памяти

```
def memory_graph(sizes, l, dl, dlr, dlrh, save):
    fig = plt.figure(figsize=(10, 7))
    plot = fig.add_subplot()
    plot.plot(sizes, l, label="Матричный Левенштейн",
        marker="o")
    plot.plot(sizes, dl, label="Матричный Дамерау–Левенштейн",
        marker="^")
    plot.plot(sizes, dlr, label="Рекурсивный Дамерау–Левенштейн
        без кэша", marker="s")
    plot.plot(sizes, dlrh, label="Рекурсивный Дамерау
       –Левенштейн с кэшем", marker="*")

    plt.legend()
    plt.grid()
    plt.title("Характеристики памяти")
    plt.ylabel("Память (байты)")
    plt.xlabel("Длина")

    if save:
        plt.savefig("memory.svg")
    else:
        plt.show()
```

Листинг 3.10 – Функция получения графика затраченного времени

```
def time_graph(sizes, l, dl, dlrh, save):
    fig = plt.figure(figsize=(10, 7))
    plot = fig.add_subplot()
    plot.plot(sizes, l, label="Матричный Левенштейн",
        marker="o")
    plot.plot(sizes, dl, label="Матричный Дамерау–Левенштейн",
        marker="^")
    plot.plot(sizes, dlrh, label="Рекурсивный Дамерау
       –Левенштейн с кэшем", marker="s")

    plt.legend()
    plt.grid()
    plt.title("Характеристики времени")
    plt.ylabel("Время (с)")
    plt.xlabel("Длина")

```

```
14      if save:
15          plt.savefig("time.svg")
16      else:
17          plt.show()
```

Листинг 3.11 – Функция получения таблицы замеров времени

```
1 def time_tables(sizes, l, dl, dlrh):
2     with open("time.csv", "w") as f:
3         writer = csv.writer(f)
4         writer.writerow(["size", "l", "dl", "dlr", "dlrh"])
5         for i in range(len(sizes)):
6             writer.writerow([f"{sizes[i]:>}", f"{l[i]: >0.5f}",
                f"{dl[i]: >0.5f}", "-", f"{dlrh[i]: >0.5f}"])
```

Листинг 3.12 – Функция получения таблицы замеров памяти

```
1 def memory_tables(sizes, l, dl, dlr, dlrh):
2     with open("memory.csv", "w") as f:
3         writer = csv.writer(f)
4         writer.writerow(["size", "l", "dl", "dlr", "dlrh"])
5         for i in range(len(sizes)):
6             writer.writerow([f"{sizes[i]:>}", f"{l[i]:>}",
                f"{dl[i]:>}", f"{dlr[i]:>}", f"{dlrh[i]:>}"])
```

Листинг 3.13 – Функция замера времени, создания таблиц и графиков и вывода результатов на экран

```
1  def test(make_tables=False):
2      time_lev = []
3      time_dam_lev = []
4      time_dam_lev_rec = []
5      time_dam_lev_rec_hash = []
6
7      memory_lev = []
8      memory_dam_lev = []
9      memory_dam_lev_rec = []
10     memory_dam_lev_rec_hash = []
11
12     n = [_ for _ in range(1, END_RANGE + 2, STEP)]
13     for i in range(1, len(n)):
14         n[i] -= 1
15
```

```
16      i = 0
17      for num in n:
18          print(f"N = {n[i]}")
19
20          s = get_random_string(num)
21          mat = [[0] * num for _ in range(num)]
22          python_link = 8
23          mat_size = getsizeof(mat)
24          s_size = getsizeof(s)
25          int_size = getsizeof(1)
26
27          time_lev.append(get_process_time(lev, num))
28          memory_lev.append(2 * s_size + 2 * int_size + mat_size)
29          print("Матричный алгоритм Левенштейна:")
30          print(f"Время: {time_lev[i]}")
31          print(f"Память: {memory_lev[i]}")
32
33          time_dam_lev.append(get_process_time(dam_lev, num))
34          memory_dam_lev.append(2 * s_size + 2 * int_size +
                mat_size)
35          print("Матричный алгоритм Дамерау–Левенштейна:")
36          print(f"Время: {time_dam_lev[i]}")
37          print(f"Память: {memory_dam_lev[i]}")
38
39          # time_dam_lev_rec.append(get_process_time(dam_lev_rec,
                num))
40          memory_dam_lev_rec.append(2 * num * (2 * int_size + 2 *
                python_link) + 2 * s_size)
41          print("Рекурсивный алгоритм Дамерау–Левенштейна без кэша
                :")
42          # print(f"Время: {time_dam_lev_rec[i]}")
43          print(f"Память: {memory_dam_lev_rec[i]}")
44
45          time_dam_lev_rec_hash.append(
46              get_process_time(dam_lev_rec_hash, num))
47          memory_dam_lev_rec_hash.append(2 * num * (2 * int_size
              + 2 * python_link) + 2 * s_size + mat_size)
48          print("Рекурсивный алгоритм Дамерау–Левенштейна с хэшем
                :")
49          print(f"Время: {time_dam_lev_rec_hash[i]}")
50          print(f"Память: {memory_dam_lev_rec_hash[i]}")
```

```
51
52          i += 1
53
54      if make_tables:
55          memory_tables(n, memory_lev, memory_dam_lev,
                memory_dam_lev_rec, memory_dam_lev_rec_hash)
56          time_tables(n, time_lev, time_dam_lev,
                time_dam_lev_rec_hash)
57
58      memory_graph(n, memory_lev, memory_dam_lev,
            memory_dam_lev_rec, memory_dam_lev_rec_hash, make_tables)
59      time_graph(n, time_lev, time_dam_lev,
            time_dam_lev_rec_hash, make_tables)
```

3.4 Функциональные тесты

В таблице 3.1 приведены функциональные тесты для алгоритмов вычисления расстояний Левенштейна и Дамерау—Левенштейна. Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты

Входные данные		**Расстояние и алгоритм**			
Строка 1	**Строка 2**	**Левенштейна**	**Дамерау-Левенштейна**		
		Итеративный	**Итеративный**	**Рекурсивный**	
				Без кеша	**С кешом**
а	б	1	1	1	1
а	а	0	0	0	0
кот	скат	2	2	2	2
капот	поток	4	4	4	4
куртка	крутка	2	1	1	1
телефон	телепорт	3	3	3	3
кот	коты	1	1	1	1

Вывод

Были реализованы алгоритмы поиска расстояния Левенштейна итеративно, а также поиска расстояния Дамерау–Левенштейна итеративно, рекурсивно и рекурсивного с кэшированием. Проведено тестирование реализаций алгоритмов.

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры по времени, представлены далее.

— Процессор: AMD Ryzen 7 3700X 8-ми ядерный процессор 3.60 Гц [4].

— Оперативная память: 16 ГБайт.

— Операционная система: Windows 10 Pro 64-разрядная система версии 22H2 [5].

При замерах времени компьютер был включен в сеть электропитания и был нагружен только системными приложениями.

4.2 Демонстрация работы программы

На рисунке 4.1 представлена демонстрация работы разработанного программного обеспечения, а именно показаны результаты вычислений реализаций алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна на примере двух строк «skat» и «kot». При этом выводятся матрицы для алгоритмов, использующих матрицы для промежуточных результатов.

```
Меню:
0) Выйти
1) Посчитать алгоритмом Левенштейна с печатанием таблицы
2) Посчитать матричным алгоритмом Дамерау-Левенштейна с печатанием таблицы
3) Посчитать рексурсивным алгоритмом Дамерау-Левенштейна
4) Посчитать рекурсивным с хэшем алгоритмом Дамерау-Левенштейна с печатанием таблицы
5) Сравнить алгоритмы
Введите номер команды: 1
Введите первую стороку: skat
Введите вторую стороку: kot
Результат:
    -   k   o   t
-   0   1   2   3
s   1   1   2   3
k   2   1   2   3
a   3   2   2   3
t   4   3   3   2
2
Меню:
0) Выйти
1) Посчитать алгоритмом Левенштейна с печатанием таблицы
2) Посчитать матричным алгоритмом Дамерау-Левенштейна с печатанием таблицы
3) Посчитать рексурсивным алгоритмом Дамерау-Левенштейна
4) Посчитать рекурсивным с хэшем алгоритмом Дамерау-Левенштейна с печатанием таблицы
5) Сравнить алгоритмы
Введите номер команды: 3
Введите первую стороку: skat
Введите вторую стороку: kot
Результат: 2
```

Рисунок 4.1 – Демонстрация работы программы при поиске расстояний Левенштейна и Дамерау-Левенштейна

4.3 Временные характеристики

Результаты эксперимента замеров по времени приведены в таблице 4.1, в которой есть поля, обозначенные «-». Это обусловлено тем, что для рекурсивной реализации алгоритмов достаточно приведенных замеров для построения графика. По полученным замерам по времени для рекурсивной реализации алгоритма Дамерау-Левенштейна без кэша понятно, что проведения замеров на длин строк больше 20 будет достаточно долгим, поэтому нет смысла проводить замеры по времени для этого алгоритма при длине строк больше 20.

Замеры проводились на одинаковых длин строк от 1 до 200 с шагом 20.

Таблица 4.1 – Замер по времени для строк, размер которых от 1 до 200

Длина (символ)	**Время, нс**			
	Левенштейн	**Дамерау-Левенштейн**		
	Итеративный	**Итеративный**	**Рекурсивный**	
			Без кэша	**С кэшем**
1	0.00000	0.00000	0.00000	0.00063
20	0.00016	0.00031	34.29323	0.00044
40	0.00047	0.00094	-	0.00119
60	0.00141	0.00172	-	0.00222
80	0.00250	0.00313	-	0.00350
100	0.00391	0.00500	-	0.00525
120	0.00594	0.00719	-	0.00719
140	0.00813	0.00969	-	0.01031
160	0.01047	0.01266	-	0.01278
180	0.01344	0.01562	-	0.01575
200	0.01578	0.01906	-	0.01906

На графике не будет отображаться время работы рекурсивного алгоритма Дамерау-Левенштейна из-за того, что значения времени во много раз превосходят показания времен других алгоритмов, и эти значения будут препятствовать информативной состовляющей графика при определении самого быстрого алгоритма.

Согласно графику 4.2, самым быстрым алгоритмом оказался нерекурсивный матричный алгоритм Левенштейна, а самым медленным из сравниваемых - рекурсивный алгоритм Дамерау-Левенштейна с хэшем.

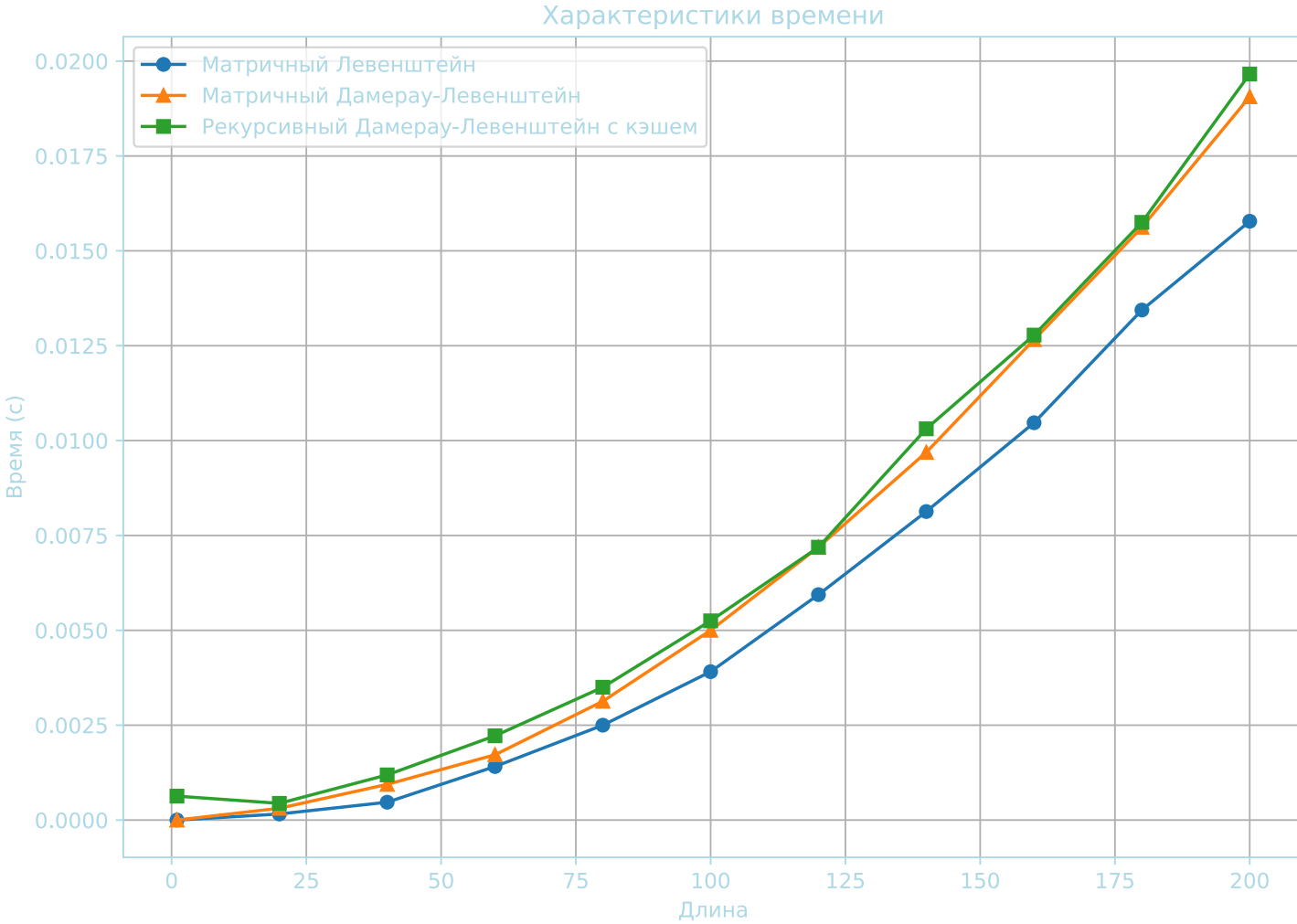

Рисунок 4.2 – Сравнение по времени реализаций алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна

Также стоит отметить, что все, представленные на графике 4.2 алгоритмы работают примерно за одно время.

4.4 Характеристики по памяти

Введем следующие обозначения:

— n — длина строки S_1;

— m — длина строки S_2;

— $size()$ — функция вычисляющая размер в байтах;

— $mat\_size(n, m)$ — функция, вычисляющая размер матрицы (n x m) типа int в байтах

— $string$ — строковый тип;

— int — целочисленный тип;

— $link\_size()$ — функция, вычисляющая размер ссылки в байтах;

Максимальная глубина стека вызовов при рекурсивной реализации нахождения расстояния Дамерау-Левенштейна равна сумме входящих строк, а на каждый вызов требуется 2 дополнительные переменные, соответственно, максимальный расход памяти равен:

$$(n + m) \cdot ((2 \cdot size(string) + 2 \cdot size(int) + 2 \cdot link\_size(string)), \quad (4.1)$$

где:

— $2 \cdot size(string)$ — хранение двух строк;

— $2 \cdot size(int)$ — хранение размеров строк;

— $2 \cdot link\_size(string)$ — хравнение ссылок на строки;

Для рекурсивного алгоритма с кэшированием поиска расстояния Дамерау-Левенштейна будет теоретически схож с расчетом в формуле (4.1), но также учитывается матрица, соответственно, максимальный расход памяти равен:

$$\begin{aligned}(n + m) \cdot (2 \cdot size(string) + 2 \cdot size(int) + 2 \cdot link\_size(string)) + \\ + mat\_size(n + 1, m + 1),\end{aligned} \quad (4.2)$$

где:

— $2 \cdot size(string)$ — хранение двух строк;

— $2 \cdot size(int)$ — хранение размеров строк;

— $2 \cdot link\_size(string)$ — хравнение ссылок на строки;

— $mat\_size(n+1, m+1)$ — хранение матрицы;

Использование памяти при итеративной реализации алгоритма поиска расстояния Левенштейна теоретически равно:

$$mat\_size(n+1, m+1) + 2 \cdot size(string) + 2 \cdot size(int), \tag{4.3}$$

где

— $2 \cdot size(string)$ — хранение двух строк;

— $2 \cdot size(int)$ — хранение размеров матрицы;

— $mat\_size(n+1, m+1)$ — хранение матрицы;

Использование памяти при итеративной реализации алгоритма поиска расстояния Дамерау-Левенштейна теоретически равно:

$$mat\_size(n+1, m+1) + 2 \cdot size(string) + 2 \cdot size(int), \tag{4.4}$$

где

— $2 * size(string)$ — хранение двух строк;

— $2 \cdot size(int)$ — хранение размеров матрицы;

— $mat\_size(n+1, m+1)$ — хранение матрицы;

По расходу памяти итеративные алгоритмы проигрывают рекурсивному алгоритму Дамерау-Левенштейна без кэша: максимальный размер используемой памяти в итеративном растет как произведение длин строк, в то время как у рекурсивного алгоритма — как сумма длин строк.

По формулам 4.1 – 4.3 затрат по памяти в программе были написаны соответствующие функции для подсчета расходуемой памяти,

результаты расчетов, которых представлены в таблице 4.2, где размеры строк находятся в диапазоне от 1 до 200 с шагом 20.

Таблица 4.2 – Замер памяти для строк, размером от 1 до 200

Длина (символ)	**Размер в байтах**			
	Левенштейн	**Дамерау-Левенштейн**		
	Итеративный	**Итеративный**	**Рекурсивный**	
			Без кэша	**С кэшем**
1	244	244	244	332
20	442	442	3018	3266
40	610	610	5938	6314
60	842	842	8858	9426
80	1106	1106	11778	12570
100	1274	1274	14698	15618
120	1474	1474	17618	18698
140	1674	1674	20538	21778
160	1906	1906	23458	24890
180	2170	2170	26378	28034
200	2210	2210	29298	30954

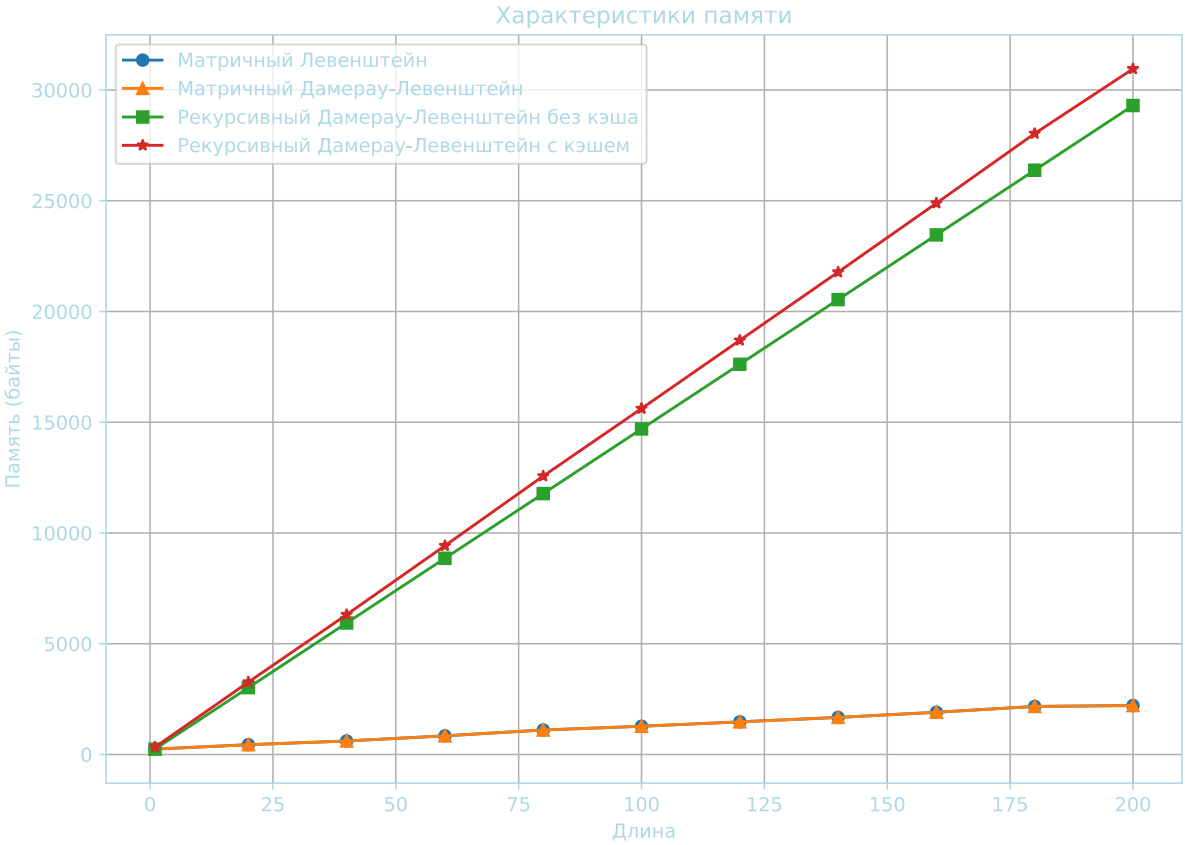

Рисунок 4.3 – Сравнение по памяти алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна

Из графика 4.3 понятно, что нерекурсивные алгоритмы используют одинаковое количество памяти. Также рекурсивные алгоритмы Дамерау-Левенштейна используют примерно одинаковое количество памяти.

4.5 Обоснования правильности замеров времени

При сравнении времени работ табличных алгоритмов Левенштейна и Дамерау-Левенштейна можно утверждать, что алгоритм Дамерау-Левенштейна работает дольше, так как в нем есть дополнительная операция проверки на транспозицию, которая замедляет его.

Если сравнить времена работы рекурсивных алгоритмов Дамерау-Левенштейна, то алгоритм с хэшем будет быстрее, так как не будет вызова рекурсий, которые были просчитаны несколько шагов ранее.

При сравнении времени работы рекурсивного алгоритма Дамерау-Левенштейна с хэшем и табличного алгоритма Дамерау-Левенштейна стоит отметить, что в табличном алгоритме, как и в рекурсивном, каждая из ячеек матрицы заполняется один раз, но в нем отсутствуют рекурсивные вызовы, что ускоряет его работу.

Таким образом можно расположить алгоритмы в порядке возрастания времени их работ: табличный алгоритм Левенштейна, табличный алгоритм Дамерау-Левенштейна, рекурсивный алгоритм Дамерау-Левенштейна без хэша, рекурсивный алгоритм Дамерау-Левенштейна с хэшем. Данные ожидания соответствуют полученному графику 4.2.

4.6 Вывод

В данном разделе было произведено сравнение количества затраченного времени и памяти алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна. Наименее затратным по времени оказался итеративный алгоритм нахождения расстояния Левенштейна.

Приведенные характеристики показывают, что рекурсивная реализация алгоритма без кэша существенно проигрывает по времени остальным алгоритмам. Данный алгоритм следует использовать при количестве символов в строке не более 1.

Так как во время печати очень часто возникают ошибки связанные с транспозицией букв [1], алгоритм поиска расстояния Дамерау-Левенштейна является наиболее предпочтительным, не смотря на то, что он проигрывает по времени и памяти алгоритму Левенштейна.

Рекурсивная реализация алгоритма поиска расстояния Дамерау-Левенштейна будет незначительно более затратным по времени по сравнению с итеративной реализацией алгоритма поиска расстояния Дамерау-Левенштейна, но намного более затратным по памяти.

Заключение

В результате исследования было определено, что время алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна растет в геометрической прогрессии при увеличении длин строк. Лучшие показатели по времени дает матричная реализация алгоритма нахождения расстояния Левенштейна и рекурсивная реализация алгоритма Дамерау-Левенштейна с кешем. Рекурсивные алгоритмы занимают намного больше памяти, чем матрицные алгоритмы.

Цель данной лабораторной работы были достигнуты, а именно описание и исследование особенностей задач динамического программирования на алгоритмах Левенштейна и Дамерау-Левенштейна.

Для достижения поставленной целей были выполнены следующие задачи.

1) Описаны алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна;

2) Создано программное обеспечение, реализующее следующие алгоритмы.

   — нерекурсивный метод поиска расстояния Левенштейна;

   — нерекурсивный метод поиска расстояния Дамерау-Левенштейна;

   — рекурсивный метод поиска расстояния Дамерау-Левенштейна;

   — рекурсивный с кешированием метод поиска расстояния Дамерау-Левенштейна.

3) Выбраны инструменты для замера процессорного времени выполнения реализаций алгоритмов.

4) Проведены анализ затрат работы программы по времени и по памяти, выяснить влияющие на них характеристики.

Список использованных источников

1 В. Ульянов М. Ресурсно-эффективные компьютерные алгоритмы: учебное пособие. — М.: Издательство «Наука», ФИЗМАТЛИ, 2007.

2 И. Левенштейн В. Двоичные коды с исправлением выпадений, вставок и замещений символов. — М.: Издательство «Наука», Доклады АН СССР, 1965. Т. 163.

3 Документация по Python [Электронный ресурс]. — Режим доступа: `https://docs.python.org/3/` (дата обращения: 10.09.2023).

4 Intel [Электронный ресурс]. — Режим доступа: `https://www.amd.com/en/processors/ryzen` (дата обращения: 10.09.2023).

5 Windows 10 Pro 2h22 64-bit [Электронный ресурс]. — Режим доступа: `https://www.microsoft.com/ru-ru/software-download/windows10` (дата обращения: 10.09.2023).