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

Отчёт по лабораторной работе №1 по дисциплине "Анализ алгоритмов"

Тема Расстояние Левенштейна

Студент Жабин Д.В.

Группа ИУ7-54Б

Преподаватель Волкова Л.Л.

Москва, 2021 г.

Оглавление

Введение

Расстояние Левенштейна [1] — метрика, измеряющая по модулю разность между двумя последовательностями символов. Она определяется как минимальное количество односимвольных операций (а именно вставки, удаления, замены), необходимых для превращения одной последовательности символов в другую.

Расстояние Левенштейна применяется в теории информации и компьютерной лингвистике для:

- поиска опечаток в запросах поисковых систем;
- автоисправления ошибок в словах;
- сравнения текстовых файлов утилитой diff;
- сравнения генов, хромосом и белков в биоинформатике.

Целью данной работы является исследование различных вариаций алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

Для достижения поставленной цели необходимо решить следующие задачи:

- изучить и реализовать алгоритмы Левенштейна и Дамерау-Левенштейна;
- протестировать реализации алгоритмов;
- провести сравнительный анализ реализаций алгоритмов нахождения расстояния между строками с точки зрения затрачиваемых ресурсов (времени и памяти).

1 Аналитическая часть

Расстояние Левенштейна между двумя строками — это минимальное количество операций вставки, удаления и замены символов, необходимых для трансформации одной строки в другую.

Цены операций могут зависеть от вида операции (перечисленных выше) и/или от участвующих в ней символов, отражая разную вероятность разных ошибок при вводе текста, и т. п.

Обозначение редакторских операций:

- I - вставка символа;
- R - замена символа;
- D - удаление символа;
- M - бездействие (применяется при совпадении символов);
- T - перестановка рядом стоящих символов (в алгоритме Дамерау - Левенштейна).

При этом для каждой операции задаётся своя цена (или штраф). Для решения задачи необходимо найти последовательность операций, минимизирующую суммарную цену всех проведённых операций. При этом следует отметить, что:

- $price(x, x) = 0$ - цена замены символа x на самого себя;
- $price(x, y) = 1$, где $(x \neq y)$ - цена замены символа x на символ y;
- $price(\varnothing, x) = 1$ - цена вставки символа x;
- $price(x, \varnothing) = 1$ - цена удаления символа x.

1.1 Рекурсивный алгоритм нахождения расстояния Левенштейна

Расстояние Левенштейна между двумя строками a и b может быть вычислено по формуле (1.1), где $a[i]$ — обозначает i-ый символ строки a.

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

А функция $m(a[i], b[j])$ определена в (1.2).

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

При этом очевидны следующие факты:

- $D(s_1, s_2) \geq ||s_1| - |s_2||$;
- $D(s_1, s_2) \leq max(|s_1|, |s_2|)$;
- $D(s_1, s_2) = 0 \Leftrightarrow s_1 = s_2$.

Функция D составлена на основе следующих утверждений ($|a|$ - длина строки a):

- для перевода из пустой строки в пустую требуется ноль операций;
- для перевода из пустой строки в строку a требуется $|a|$ операций;
- для перевода из строки a в пустую требуется $|a|$ операций;
- для перевода из строки a в строку b требуется выполнить определенное количество операций (удаление, вставка, замена) в некоторой последовательности. Последовательность проведения любых двух операций можно поменять, порядок проведения операций не имеет никакого значения. Полагая, что a', b' — строки a и b без последнего символа соответственно, цена преобразования из строки a в строку b может быть выражена так:

- 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 разные символы;
- цена преобразования из a' в b', предполагая, что a и b оканчиваются на один и тот же символ.

Минимальной ценой преобразования будет минимальное значение среди приведенных вариантов.

Рекурсивный алгоритм представляет собой реализацию формулы (1.1).

1.2 Нерекурсивный алгоритм нахождения расстояния Левенштейна с кэшем в форме двух строк

Можно заметить, что прямая реализация формулы (1.1) оказывается неэффективной для больших i, j, так как многие промежуточные значения $D(i, j)$ неоднократно вычисляются повторно. Можно оптимизировать алгоритм нахождения расстояния Левенштейна, используя дополнительную матрицу в целях хранения соответствующих промежуточных значений. В таком случае алгоритм представляет собой построчное заполнение матрицы $A_{|a|,|b|}$ значениями $D(i, j)$.

Обратив внимание на процесс заполнения ячеек матрицы, можно заметить, что на каждом шаге используются только две последние строки матрицы, что приводит к идее использовать кэш в формате двух строк, в результате чего сократится объем потребляемой памяти.

1.3 Рекурсивный алгоритм нахождения расстояния Левенштейна с кэшем в форме матрицы

Этот метод совмещает в себе алгоритмы (1.1) и (1.2) — происходит параллельное заполнение 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.

1.4 Алгоритм нахождения расстояния Дамерау–Левенштейна

При нахождении расстояния Дамерау–Левенштейна добавляется операция транспозиции (перестановки соседних символов).

Расстояние Дамерау–Левенштейна может быть найдено по формуле (1.3).

$$D(i,j) = \begin{cases} 0, & \text{i = 0, j = 0;} \\ i, & \text{j = 0, i > 0;} \\ j, & \text{i = 0, j > 0;} \\ \min\{ & \\ \quad D(i, j-1) + 1 & \\ \quad D(i-1, j) + 1 & \text{i > 0, j > 0.} \\ \quad D(i-1, j-1) + m(a[i], b[j]) & \\ \quad \left[\begin{array}{ll} D(i-2, j-2) + 1, & \text{если } i, j > 1, \\ & a[i] = b[j-1], \\ & b[j] = a[i-1] \\ \infty, & \text{иначе} \end{array}\right. & \\ \}, & \end{cases} \tag{1.3}$$

Алгоритм, реализующий эту функцию, так же, как и алгоритм (1.1) можно оптимизировать, используя кэш-матрицу для сохранения уже вычисленных значений.

1.5 Вывод по аналитической части

В данном разделе были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна, который является модификацией первого, учитывающей возможность перестановки соседних символов, а также их вариации, использующие кэш для устранения такого недостатка, как повторное вычисление уже посчитанных значений.

2 Конструкторская часть

На основе полученных аналитических данных построим схемы алгоритмов нахождения расстояния Левенштейна и Дамерау–Левенштейна.

2.1 Схемы алгоритмов

На рисунках 2.1 – 2.4 представлены схемы алгоритмов нахождения расстояния Левенштейна (с кэшем в форме 2 строк), расстояния Левенштейна (рекурсивно, без использования кэша), расстояния Левенштейна (рекурсивно, с кэшем в форме матрицы) и расстояния Дамерау–Левенштейна соответственно.

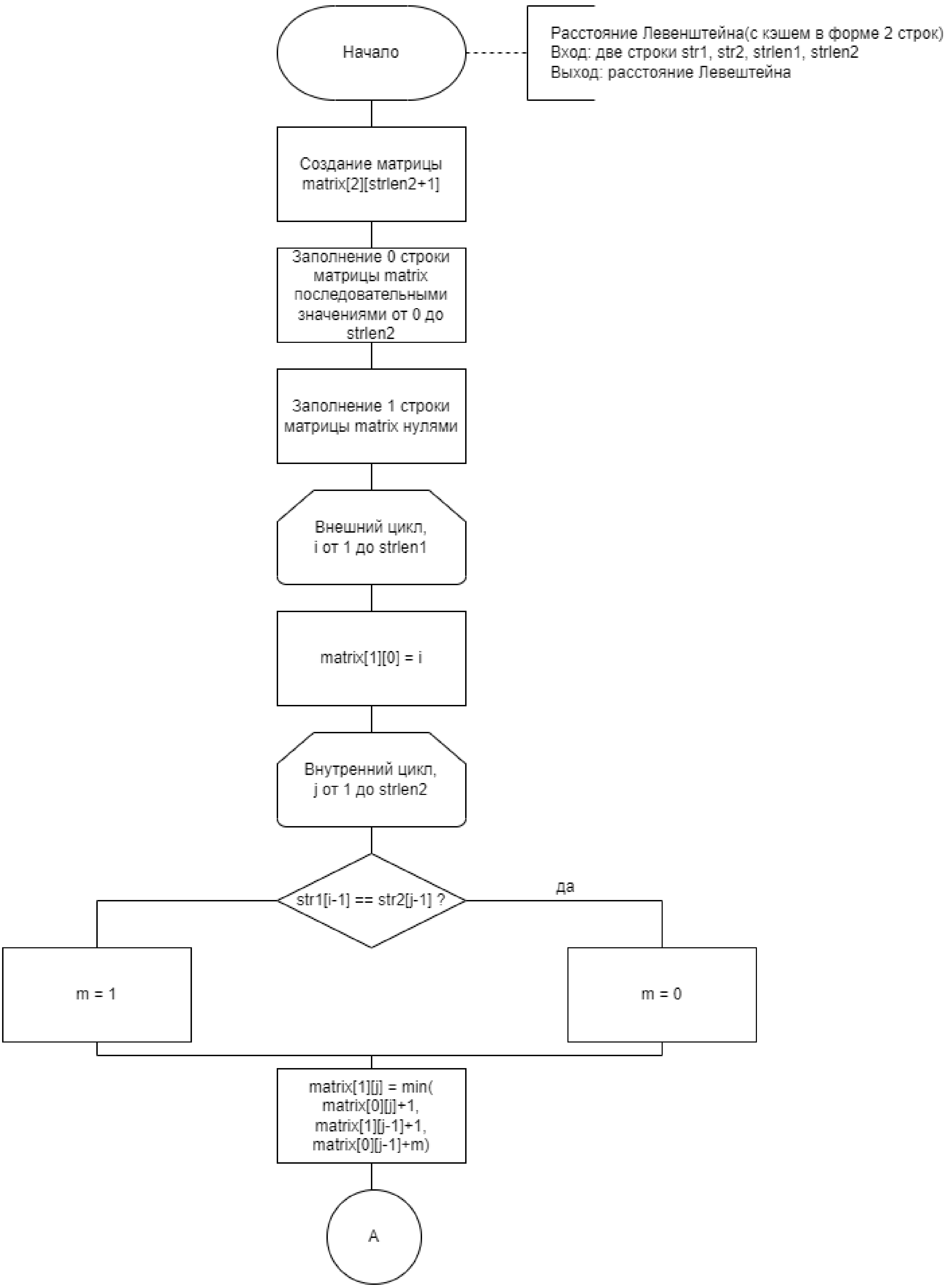

Начало
Расстояние Левенштейна(с кэшем в форме 2 строк)
Вход: две строки str1, str2, strlen1, strlen2
Выход: расстояние Левештейна
Создание матрицы matrix[2][strlen2+1]
Заполнение 0 строки матрицы matrix последовательными значениями от 0 до strlen2
Заполнение 1 строки матрицы matrix нулями
Внешний цикл, i от 1 до strlen1
matrix[1][0] = i
Внутренний цикл, j от 1 до strlen2
str1[i-1] == str2[j-1] ?
да
m = 1
m = 0
matrix[1][j] = min(matrix[0][j]+1, matrix[1][j-1]+1, matrix[0][j-1]+m)
A

Рисунок 2.1 — Расстояние Левенштейна с двухстрочным кэшем

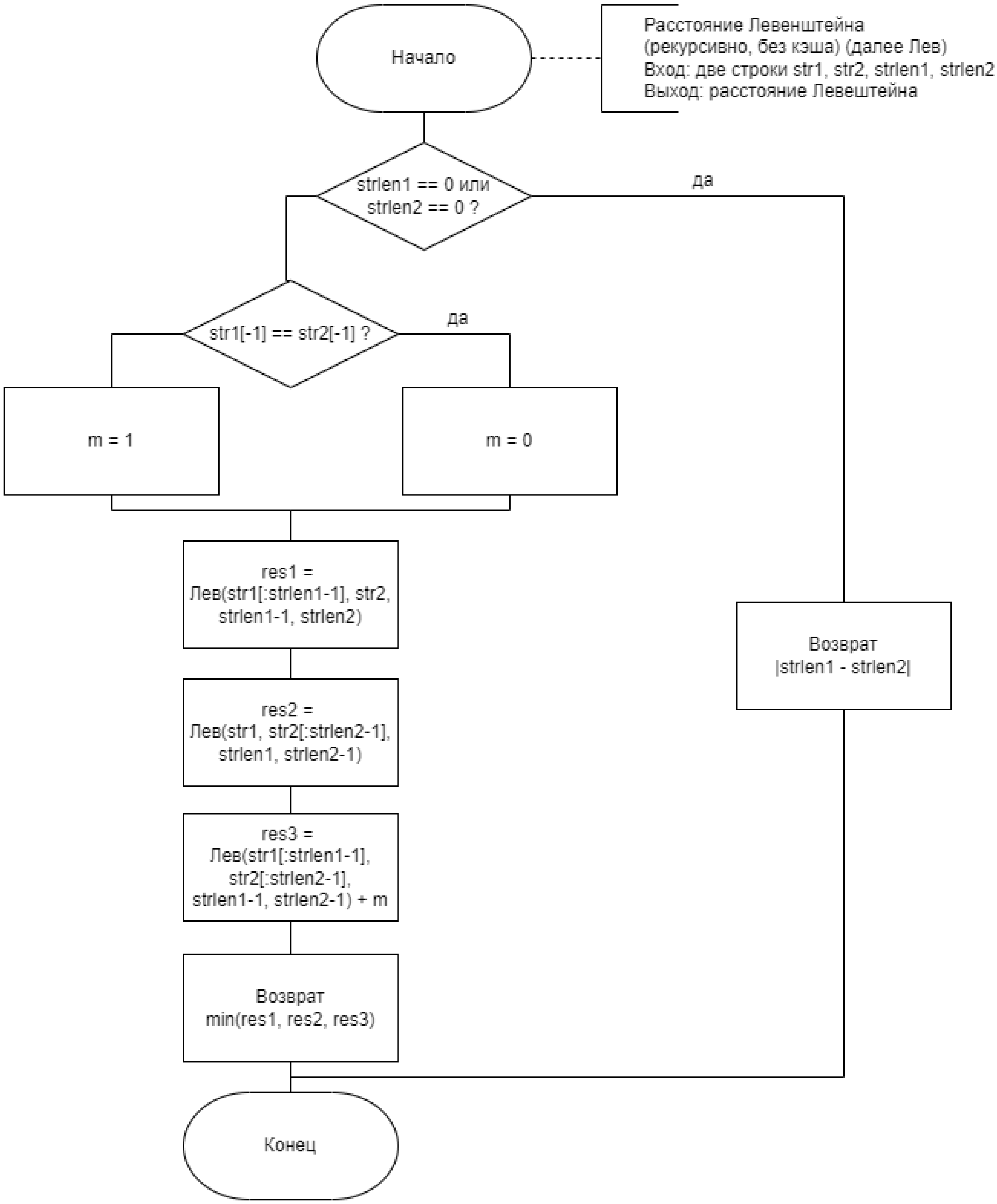
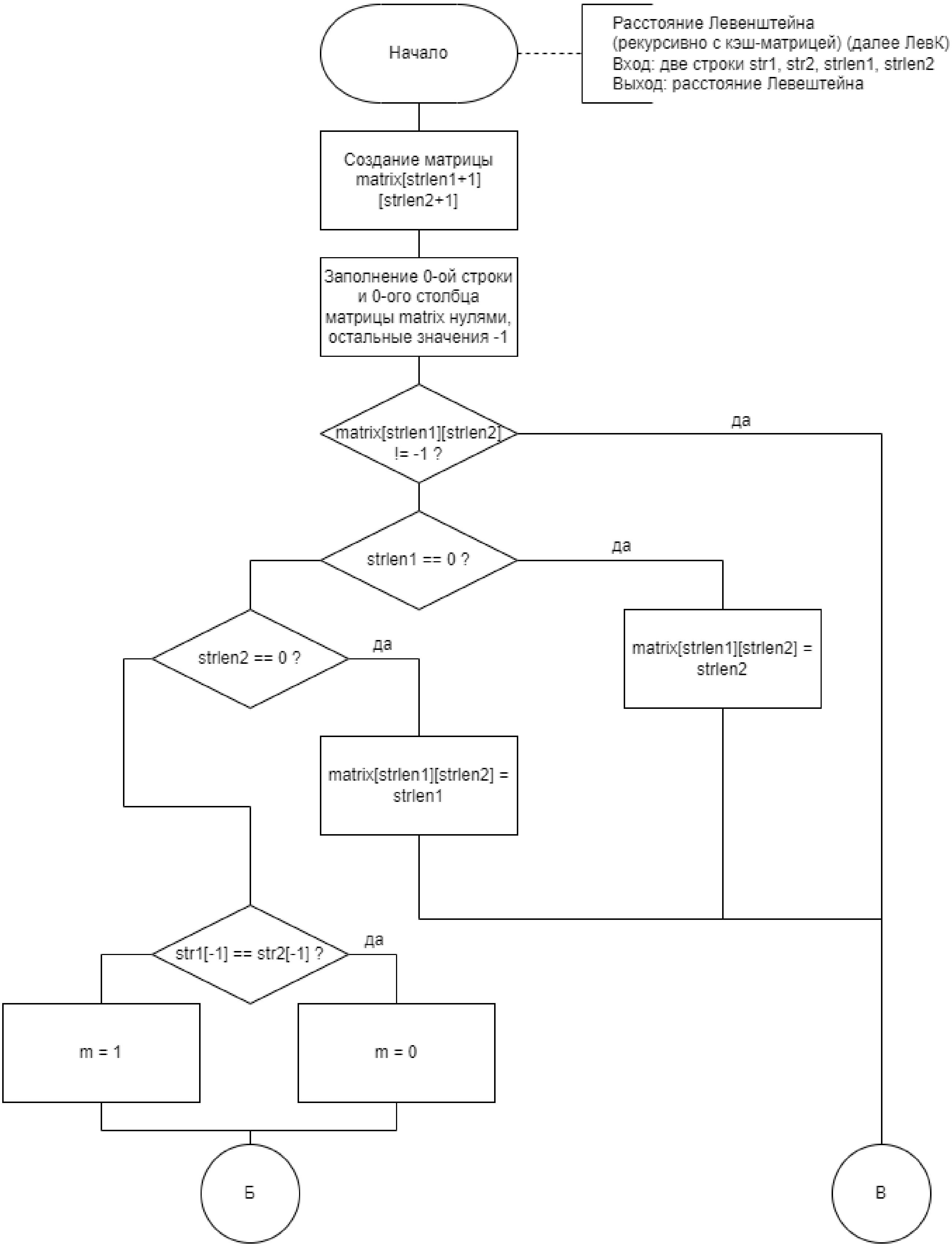


Рисунок 2.2 — Расстояние Левенштейна рекурсией без кэша


Начало
Расстояние Левенштейна
(рекурсивно с кэш-матрицей) (далее ЛевК)
Вход: две строки str1, str2, strlen1, strlen2
Выход: расстояние Левештейна
Создание матрицы matrix[strlen1+1][strlen2+1]
Заполнение 0-ой строки и 0-ого столбца матрицы matrix нулями, остальные значения -1
matrix[strlen1][strlen2] != -1 ?
да
strlen1 == 0 ?
да
matrix[strlen1][strlen2] = strlen2
strlen2 == 0 ?
да
matrix[strlen1][strlen2] = strlen1
str1[-1] == str2[-1] ?
да
m = 1
m = 0
Б
В

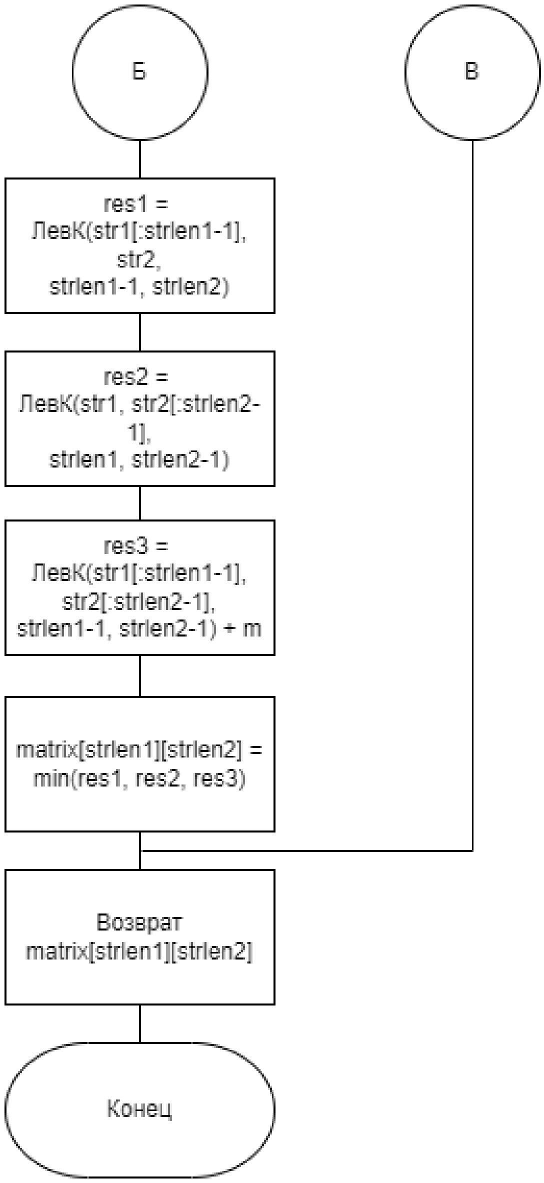


Рисунок 2.3 — Расстояние Левенштейна рекурсией с кэш-матрицей

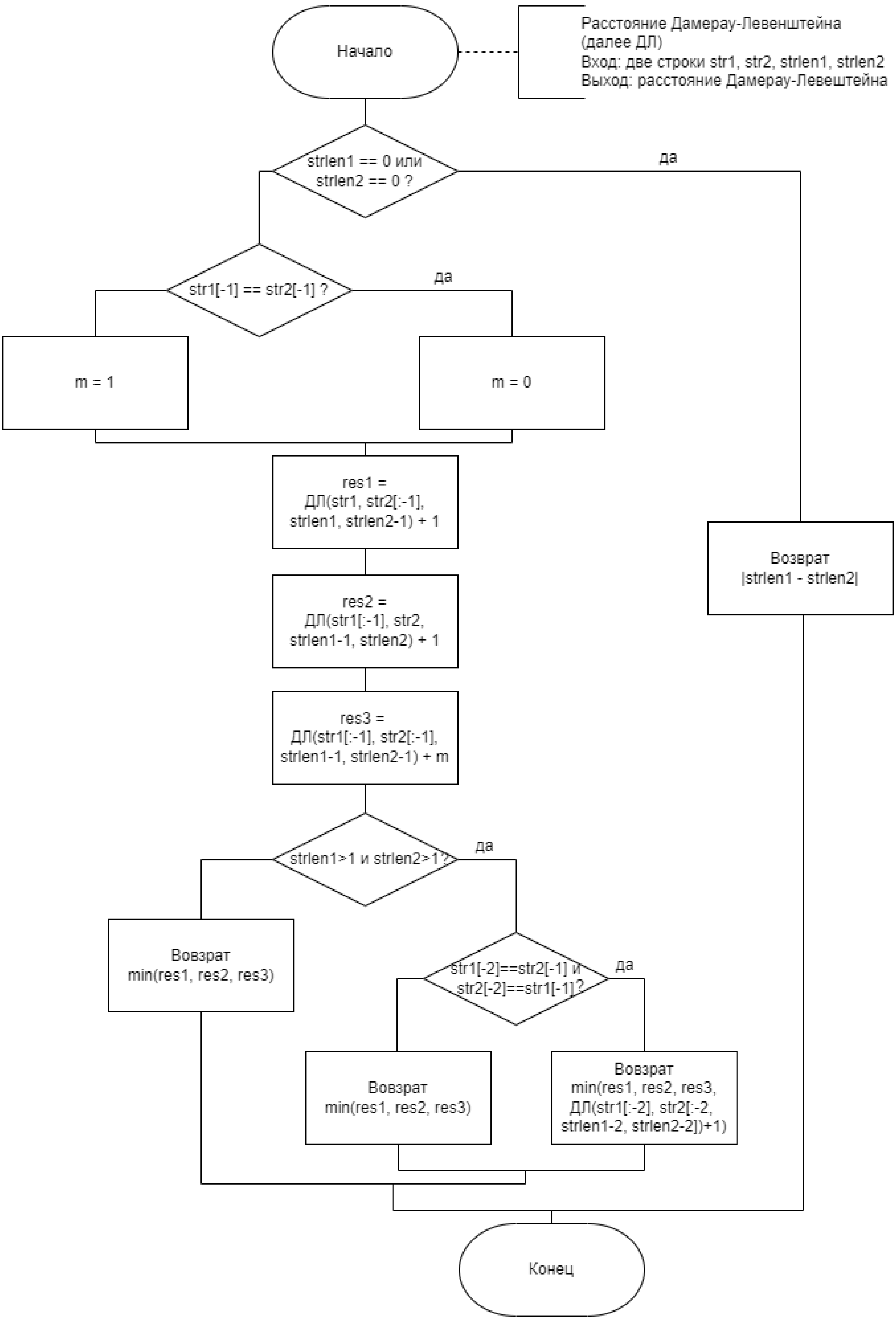


Рисунок 2.4 — Расстояние Дамерау–Левенштейна

2.2 Вывод по конструкторской части

На основе теоретических данных, полученных из аналитического раздела, были построены схемы алгоритмов нахождения расстояний Левенштейна и Дамерау–Левенштейна.

3 Технологическая часть

В данном разделе приведены средства реализации и листинги кода.

3.1 Требования к ПО

К программе предъявляется ряд требований:

- на вход ПО получает две строки;
- на выходе — расстояние Левенштейна (или Дамерау–Левенштейна) для этих строк.

3.2 Средства реализации

Для реализации ПО был выбран язык программирования `Python` [2]. Это обусловлено знанием возможностей языка, что обеспечит высокую скорость написания программы без потери ее качества.

В качестве среды разработки была выбрана `Visual Studio Code` [4]. Удобства написания кода и его автодополнения стали ключевыми при выборе.

3.3 Реализация алгоритмов

В листингах 3.1 - 3.4 приведена реализация рассматриваемых алгоритмов.

Листинг 3.1 — Расстояние Левенштейна с двухстрочным кэшем

```
def levTable(str1, str2, strlen1, strlen2):
  matrix = []
  matrix.append([])
  for j in range(strlen2+1):
    matrix[0].append(j)
  if strlen1:
    matrix.append([])
    for j in range(strlen2+1):
      matrix[1].append(0)
  for i in range(1, strlen1+1):
    matrix[1][0] = i
    for j in range(1, strlen2+1):
      matrix[1][j] = min(matrix[0][j] + 1,
          matrix[1][j - 1] + 1,
          matrix[0][j - 1] + lastEqu(str1[:i], str2[:j]))
    matrix[0] = deepcopy(matrix[1])
  return matrix[-1][-1]
```

Листинг 3.2 — Расстояние Левенштейна рекурсией без кэша

```
def levRec(str1, str2, strlen1, strlen2):
  if not str1 or not str2:
    return abs(strlen1 - strlen2)
  res = min(levRec(str1, str2[:-1], strlen1, strlen2-1) + 1,
    levRec(str1[:-1], str2, strlen1-1, strlen2) + 1,
    levRec(str1[:-1], str2[:-1], strlen1-1, strlen2-1) + lastEqu(str1, str2))
  return res
```

Листинг 3.3 — Расстояние Левенштейна рекурсией с кэш-матрицей

```
def levRecMatr(str1, str2, strlen1, strlen2):
    matrix = [[i+j if (i == 0 or j == 0) else -1 for j in
     range(strlen2 + 1)] for i in range(strlen1 + 1)]
    recursion(str1, str2, strlen1, strlen2, matrix)
    return matrix[strlen1][strlen2]

def recursion(str1, str2, i, j, matrix):
    if matrix[i][j] != -1:
        return matrix[i][j]
    if not i or not j:
        return abs(len(str1) - len(str2))
    matrix[i][j] = min(recursion(str1, str2, i - 1, j, matrix) + 1,
        recursion(str1, str2, i, j - 1, matrix) + 1,
        recursion(str1, str2, i - 1, j - 1, matrix) +
        lastEqu(str1[:i], str2[:j]))
    return matrix[i][j]
```

Листинг 3.4 — Расстояние Дамерау–Левенштейна

```
def damlevRec(str1, str2, strlen1, strlen2):
    if not str1 or not str2:
        return abs(strlen1 - strlen2)
    res = min(damlevRec(str1, str2[:-1], strlen1, strlen2-1) + 1,
        damlevRec(str1[:-1], str2, strlen1-1, strlen2) + 1,
        damlevRec(str1[:-1], str2[:-1], strlen1-1, strlen2-1) + lastEqu(str1, str2))
    if (strlen1 > 1 and strlen2 > 1 and str1[-1] == str2[-2] and str1[-2] == str2[-1]):
        res = min(res, damlevRec(str1[:-2], str2[:-2],
          strlen1-2, strlen2-2) + 1)
    return res
```

В листинге (3.5) представлена функция `lastEqu`, использующаяся в алгоритмах поиска расстояний Левенштейна и Дамерау–Левенштейна для определения совпадения последних символов в двух строках.

Листинг 3.5 — Совпадение последних символов

```
def lastEqu(str1, str2):
    if not len(str1) or not len(str2):
        return 1
    return 0 if (str1[-1] == str2[-1]) else 1
```

3.4 Тестовые данные

В таблице (3.1) приведены тесты для реализованных функций. Все тесты пройдены успешно.

Таблица 3.1 — Тестирование

Строка1	Строка2	Результат	Ожидаемый результат
$\varnothing$	$\varnothing$	0	0
$\varnothing$	*data*	4	4
data	$\varnothing$	4	4
robot	*gorod*	3	3
a	*data*	3	3
data	*a*	3	3
bigdata	*ba*	5	5
ba	*bigdata*	5	5
bigdata	*gda*	4	4
gda	*bigdata*	4	4
ibgcaat	*bigdata*	5(3[1])	5(3[1])
bigdata	*ibgcaat*	5(3[1])	5(3[1])

3.5 Вывод по технологической части

В данном разделе были разработаны и протестированы реализации алгоритмов нахождения расстояния Левенштейна и Дамерау–Левенштейна.

[1] расстояние Дамерау–Левенштейна

4 Исследовательская часть

В этом разделе будут исследованы ресурсы, затрачиваемые реализациями алгоритмов, на практических тестах.

4.1 Технические характеристики

Ниже приведены технические характеристики устройства, на котором было проведено тестирование ПО:

- Операционная система Windows 10 64-разрядная.
- Оперативная память 16 ГБ.
- Процессор Intel(R) Core(TM) i5-4690 @ 3.50ГГц.

4.2 Время выполнения реализаций алгоритмов

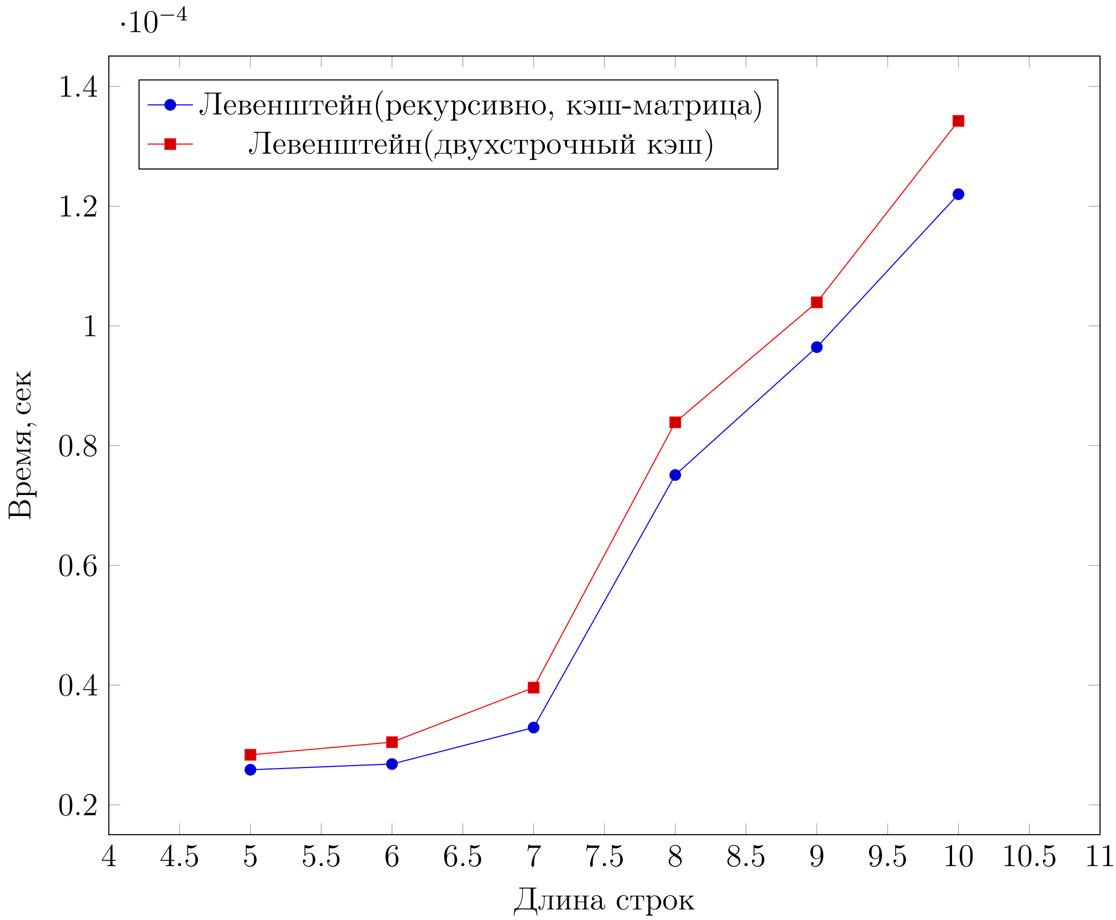
Время выполнения алгоритмов замерялось с помощью специальной функции `process_time()` [3] из модуля `time`, которая возвращает значение в долях секунды процессорного времени текущего процесса. Контрольная точка возвращаемого значения не определена, поэтому допустима только разница между результатами последовательных вызовов.

В таблице 4.1 и на графиках 4.1 - 4.2 показаны результаты замеров. Все значения времени выполнения отображены в секундах.

Таблица 4.1 — Зависимость времени выполнения реализаций алгоритмов от длины строк

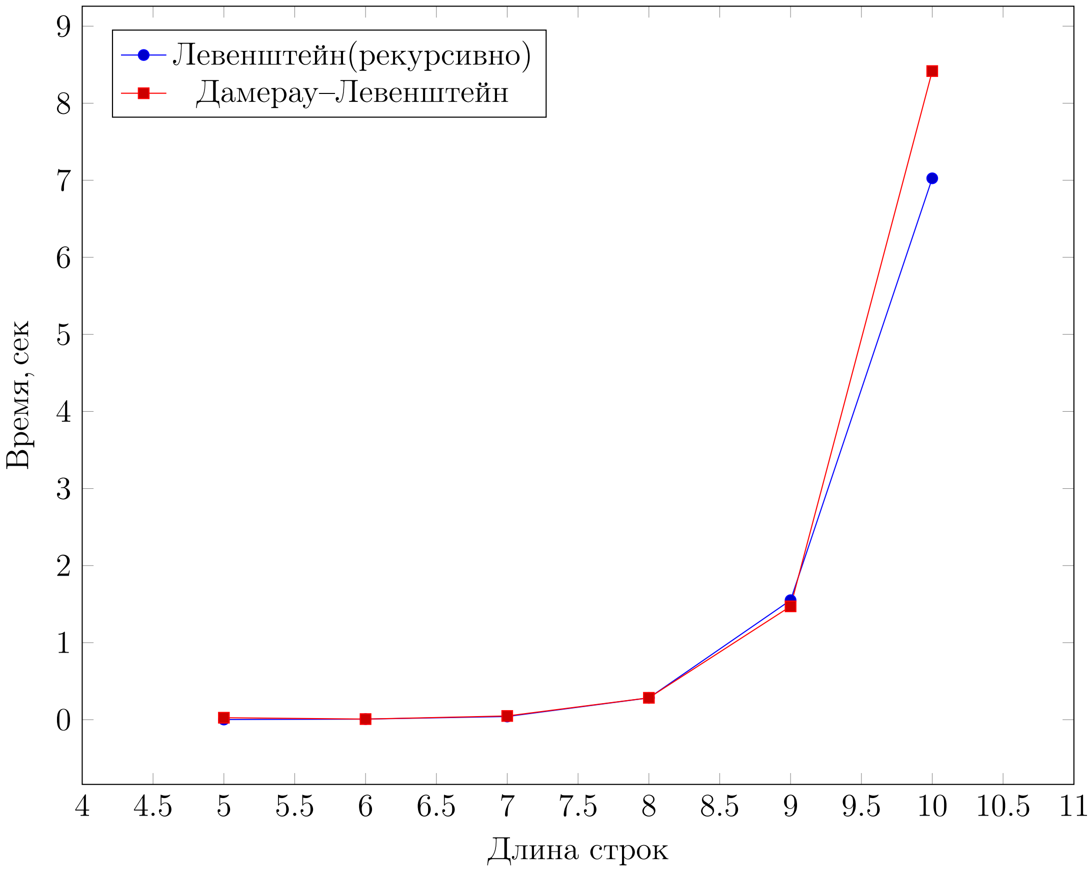
Длина строк	LevCache[1]	LevRec[2]	LevRecCache[3]	Dam–Lev[4]
5	0.00002837	0.00185848	0.00002587	0.00166622
6	0.00003048	0.00767277	0.00002683	0.00806983
7	0.00003959	0.04125901	0.00003292	0.04837457
8	0.00008390	0.28312333	0.00007510	0.28398161
9	0.00010394	1.55095931	0.00009646	1.47087310
10	0.00013424	7.02560350	0.00012200	8.41714851

График 4.1 — Зависимость времени выполнения от длины строк (алгоритмы нахождения расстояния Левенштейна рекурсивно с кэш-матрицей и итеративно с кэшем в форме 2 строк)



[1]расстояние Левенштейна с кэшем в форме 2 строк
[2]расстояние Левенштейна рекурсией без кэша
[3]расстояние Левенштейна рекурсией с кэш-матрицей
[4]расстояние Дамерау–Левенштейна

График 4.2 — Зависимость времени выполнения от длины строк (алгоритмы нахождения расстояния Левенштейна рекурсивно без кэша и Дамерау–Левенштейна)



4.3 Используемая память

Пусть $str1$, $str2$ — строки, $strlen1$ — длина строки $str1$, $strlen2$ — длина строки $str2$, $sizeof$ — операция получения размера типа данных на конкретной машине. Тогда, рассчитаем объем используемой памяти для каждого алгоритма.

Алгоритм Левенштейна с кэшем в виде двух строк

- две целочисленные переменные для хранения длин строк $2sizeof(int)$;
- память для хранения строк $(strlen1 + strlen2) \cdot sizeof(char)$;
- память для хранения кэш-матрицы $(strlen2 + 1) \cdot 2sizeof(int)$;
- память для хранения 2 целочисленных вспомогательных переменных $2 \cdot sizeof(int)$.

Суммарная память представлена в формуле (4.1).

$$(strlen1 + strlen2) \cdot sizeof(char) + (strlen2 + 3) \cdot 2sizeof(int) \tag{4.1}$$

Рекурсивный алгоритм Левенштейна

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк. Для каждого вызова:

- две целочисленные переменные для хранения длин строк $2 sizeof(int)$;
- память для хранения строк $(strlen1 + strlen2) \cdot sizeof(char)$;
- память для хранения одной целочисленной вспомогательной переменной $sizeof(int)$;
- память для хранения адреса возврата $sizeof(int*)$.

Суммарная память представлена в формуле (4.2).

$$(strlen1 + strlen2)\cdot$$
$$\cdot((strlen1 + strlen2) \cdot sizeof(char) + 3 sizeof(int) + sizeof(int*)) \tag{4.2}$$

Рекурсивный алгоритм Левенштейна с кэш-матрицей

Память для хранения кэш-матрицы $(strlen1+1)(strlen2+1)\cdot sizeof(int)$.

Для каждого вызова:

- две целочисленные переменные для хранения длин строк $2 sizeof(int)$;
- память для хранения строк$(strlen1 + strlen2) \cdot sizeof(char)$;
- память для хранения 2 целочисленных вспомогательных переменных $2 sizeof(int)$;
- память для хранения ссылки на матрицу $sizeof(int*)$;
- память для хранения адреса возврата $sizeof(int*)$.

Суммарная память представлена в формуле (4.3).

$$(strlen1 + 1) \cdot (strlen2 + 1) \cdot sizeof(int) + (strlen1 + strlen2)\cdot$$
$$\cdot((strlen1 + strlen2) \cdot sizeof(char) + 4 sizeof(int) + 2 sizeof(int*)) \tag{4.3}$$

Алгоритм Дамерау–Левенштейна

Аналогичен рекурсивному алгоритму Левенштейна без использования кэша, однако используется дополнительная целочисленная переменная. Суммарная память представлена в формуле (4.4).

$$\begin{gathered}(strlen1 + strlen2)\cdot \\ \cdot((strlen1 + strlen2) \cdot sizeof(char) + 4sizeof(int) + sizeof(int*))\end{gathered} \tag{4.4}$$

4.4 Вывод по исследовательской части

Проведены замеры времени выполнения реализаций алгоритмов и оценено количество затрачиваемой памяти.

По результатам самый быстрый алгоритм — алгоритм нахождения расстояния Левенштейна рекурсией с использованием кэш-матрицы, при этом является самым затратным с точки зрения памяти, в среднем этот алгоритм затрачивает памяти больше на размер кэш-матрицы. Значит с увеличением длин строк, он будет требовать все больше памяти.

Самым медленным алгоритмом, явно на больших длинах строк, является алгоритм нахождения расстояния Дамерау–Левенштейна.

Алгоритм нахождения расстояния Левенштейна с использованием двухстрочного кэша требует меньше всего памяти за счет отсутствия рекурсивных вызовов.

Заключение

В ходе проделанной работы была достигнута поставленная цель и решены следующие задачи:

- были изучены алгоритмы нахождения расстояний Левенштейна и Дамерау–Левенштейна;
- эти алгоритмы были реализованы и успешно протестированы;
- был проведён сравнительный анализ алгоритмов по времени работы и количеству затрачиваемой памяти.

Выбор конкретного алгоритма нахождения расстояния будет зависеть от исходных данных и поставленной задачи. Для достижения наивысшей скорости работы, при этом потери большого количества памяти следует выбрать алгоритм нахождения расстояния Левенштейна рекурсией с использованием кэш-матрицы. Самым сбалансированным вариантом является итеративный алгоритм с использованием кэша в форме двух строк.

Литература

[1] Левенштейн В.И. Двоичные коды с исправлением выпадений, вставок и замещений символов. Доклады АН СССР, 1965. Т. 163. С. 845-848.

[2] Python [Электронный ресурс]. Режим доступа: https://python.org. Дата обращения: 10.10.2021.

[3] Модуль time [Электронный ресурс]. Режим доступа: https://docs.python.org/3/library/time.html. Дата обращения: 10.10.2021.

[4] Visual Studio Code - Code Editing [Электронный ресурс]. Режим доступа: https://code.visualstudio.com. Дата обращения: 10.10.2021.