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 дисциплине: Анализ Алгоритмов

Тема: Расстояние Левенштейна и Дамерау-Левенштейна

Студент Чаушев А.К..

Группа ИУ7-56Б

Оценка (баллы) _____

Преподаватели Волкова Л.Л., Строганов Ю.В.

Москва — 2020 г.

Введение

Расстояние Левенштейна (также известное как редакционное расстояние или дистанция редактирования) в теории информации и компьютерной лингвистике – мера различия двух последовательностей символов (строк) относительно минимального количества операций вставки, удаления и замены, необходимых для перевода одной строки в другую . Для одинаковых строк расстояние редактирования равно нулю.

В 1965 году советский математик Владимир Иосифович Левенштейн разработал алгоритм, который позволяет оценить, насколько похожа одна строка на другую. Алгоритм Левенштейна дает возможность получить именно численную оценку похожести строк. Основная идея алгоритма состоит в том, чтобы посчитать минимальное количество операций удаления, вставки и замены, которые необходимо сделать над одной из строк, чтобы получить вторую.

1 Аналитическая часть

1.1 Постановка задачи

Цель данной лабораторной работы: разработать и сравнить алгоритмы поиска расстояний Ливенштейна и Демерау-Ливенштейна с использованием метода динамического программирования.
В данной работе требуется:

1) описать алгоритмы поиска Левенштейа и Демерау - Левенштейна для нахождения редакционного расстояния между строками;

2) реализовать данные алгоритмы на одном из языков программирования;

3) провести сравнительный анализ алгоритмов по затраченному времени и памяти;

4) привести пример работы всех указанных алгоритмов;

1.2 Описание растояния

1.2.1 Растояние Левенштейна

Для поиска расстояния Левенштейна, чаще всего используют алгоритм в котором необходимо заполнить матрицу D, размером n + 1 на m + 1, где n и m – длины сравниваемых строк A и B, по следующим правилам:

- $D_{0,0} = 0$;
- $D_{i,j}$ = минимальное из:
  - $D_{i-1,j-1+0}$ (символы одинаковые), либо $D_{i-1,j-1}$ + Creplace(замена символа);
  - $D_{i,j-1}$ + Cdelete (удаление символа);
  - $D_{i-1,j}$ + Cinsert (вставка символа);

Cdelete, Cinsert, Creplace – цена или вес удаления, вставки и замены символа. При этом $D_{n-1,m-1}$ – содержит значение расстояния Левенштейна.

Пусть S1 и S2 - две строки (длиной n и m соответственно) над некоторым алфавитом, тогда редакционное расстояние D(S1,S2) можно подсчитать по следующей рекуррентной формуле (1) :

$$D(S_1[i]S_2[j]) = min \begin{cases} D(S_1[1..i], S_2[1..j-1]) + 1 \\ D(S_1[1..i-1], S_2[1..j]) + 1 \\ D(S_1[1..i-1], S_2[1..j-1]) + \begin{bmatrix} 0, & \text{если } S_{1[i]} = S_2[j]. \\ 1, & \text{иначе.} \end{bmatrix} \end{cases} \quad (1)$$

Рассмотрим формулу более подробно. Здесь шаг по i символизирует удаление (D) из первой строки, по j - вставку (I) в первую строку, а шаг по обоим индексам символизирует замену символа (R) или отсутствие изменений (M). Очевидно, что редакционное расстояние между двумя пустыми строками равно нулю. Так же очевидно то, что чтобы получить пустую строку из строки длиной i, следует совершить i операций удаления, а чтобы получить строку длиной j из пустой, требуется произвести j операций вставки. В нетривиальном случае необходимо выбрать минимальную «стоимость» из трёх вариантов. Вставка/удаление будет в любом случае стоить одну операцию, а вот замена может не понадобиться, если символы равны - тогда шаг по обоим индексам бесплатный. Формализация этих рассуждений приводит к формуле, указанной выше.

1.2.2 Растояние Дамерау-Левенштейна

В автоматической обработке естественного языка (например, при автоматической проверке орфографии) часто бывает нужно определить, насколько различны два написанных слова. Одна из количественных мер, используемых для этого, называется расстоянием Дамерау–Левенштейна [3] - в честь Владимира Левенштейна и Фредерика Дамерау. Левенштейн придумал способ измерения «расстояний» между словами, а Дамерау независимо от него выделил несколько классов, в которые попадает большинство опечаток.

Эта вариация алгоритма вносит в определение расстояния Левенштейна еще одно правило — транспозиция (перестановка) двух соседних букв также учитывается как одна операция, наряду со вставками, удалениями и заменами.

Чтобы вычислять такое расстояние, достаточно немного модифицировать алгоритм нахождения обычного расстояния Левенштейна следующим образом: хранить не две, а три последних строки матрицы, а также добавить соответствующее дополнительное условие — в случае обнаружения транспозиции при расчете расстояния также учитывать и её стоимость.

$$D(S_1[i]S_2[j]) = min \begin{cases} D(S_1[1..i], S_2[1..j-1]) + 1 \\ D(S_1[1..i-1], S_2[1..j]) + 1 \\ D(S_1[1..i-1], S_2[1..j-1]) + \begin{bmatrix} 0, & \text{если } S_{1[i]} = S_2[j]. \\ 1, & \text{иначе.} \end{bmatrix} \\ D(S_1[1..i-1], S_2[1..j-1]) + \begin{bmatrix} 1, & \text{если } S_{1[i]} = S_2[j-1] \text{ и } S_{1[i]} = S_2[j-1]. \\ \infty, & \text{иначе.} \end{bmatrix} \end{cases}.$$

(2)

2 Конструкторская часть

2.1 Разработка алгоритмов

2.1.1 Схема алгоритма

На рисунках 1 - 4 показаны схемы алгоритмов Левенштейна итеративная, рекурсивная , рекурсивная реализация с заполнением матрицы и схема алгоритма Дамерау–Левенштейна.

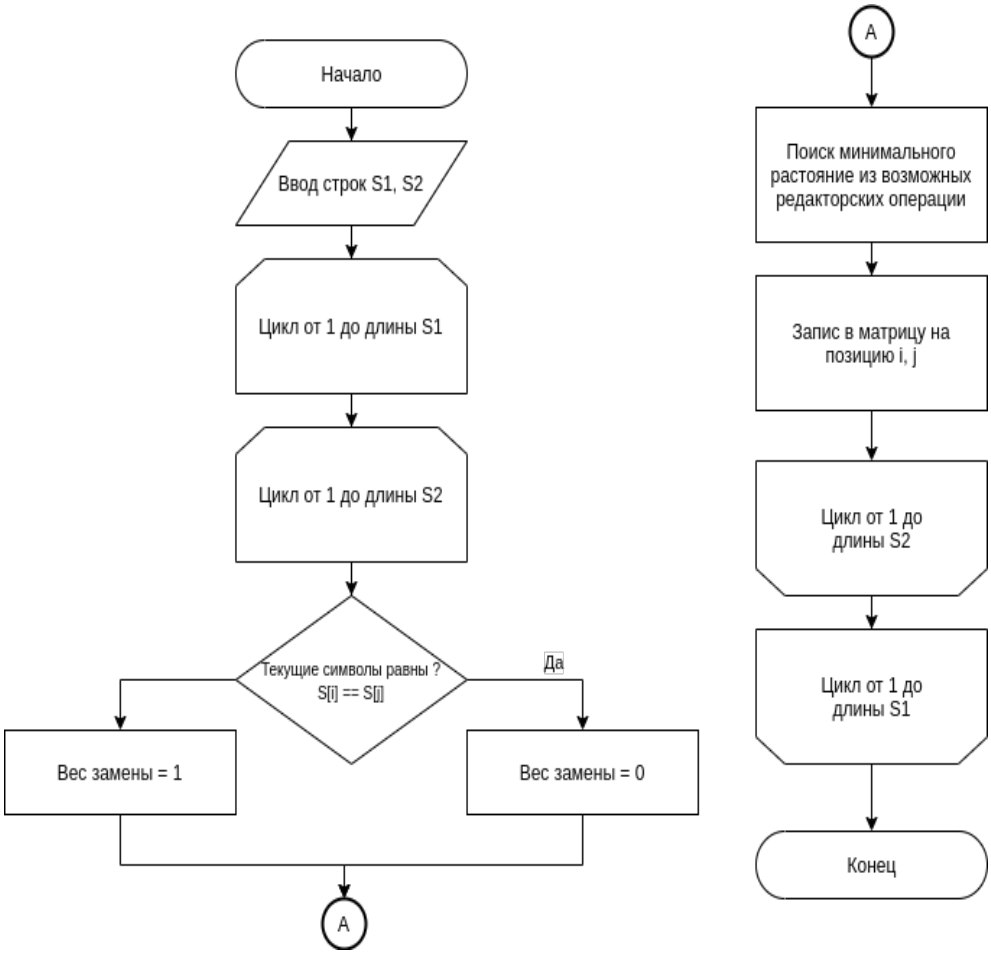

Рис. 1: Схема алгоритма нахождения расстояния Левенштейна сделана по реализации кода в приложении.

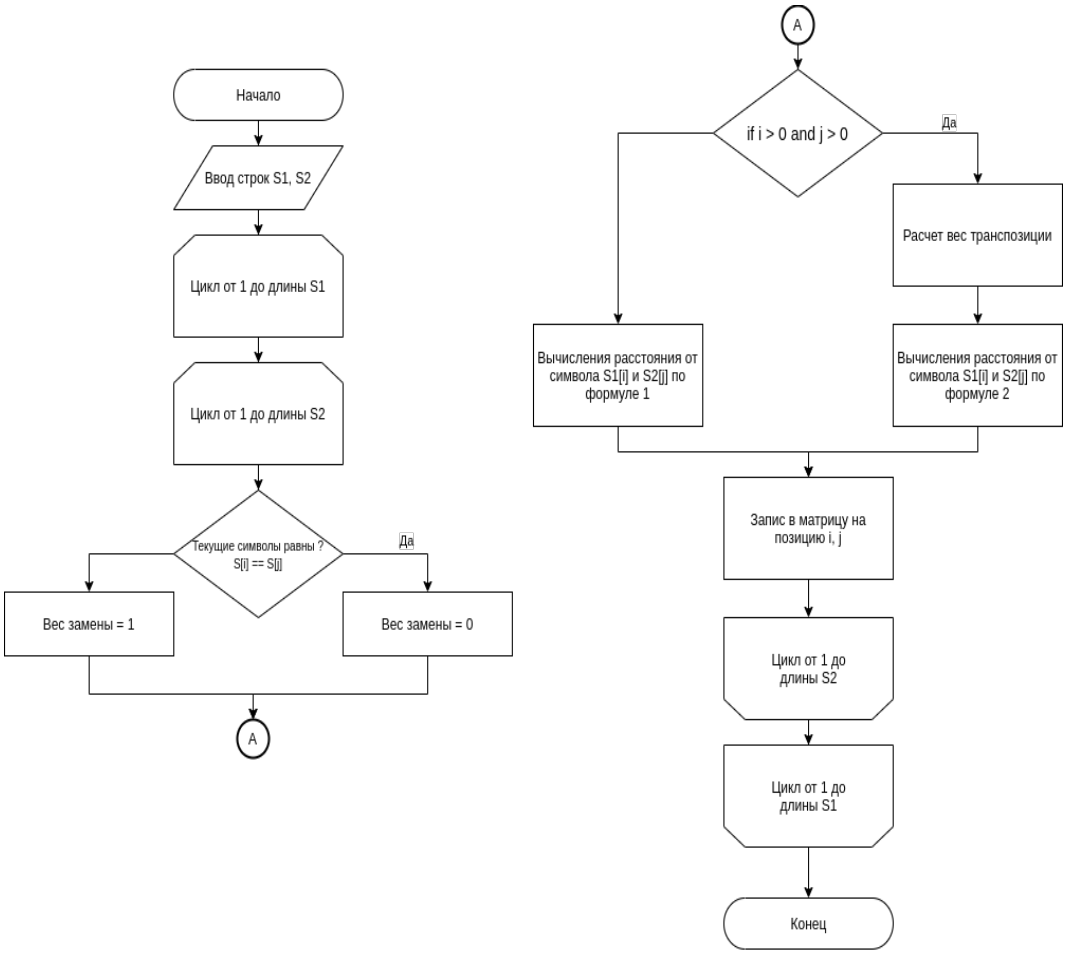

Рис. 2: Схема алгоритма нахождения расстояния Дамерау-Левенштейна сделана по реализации кода в приложении.

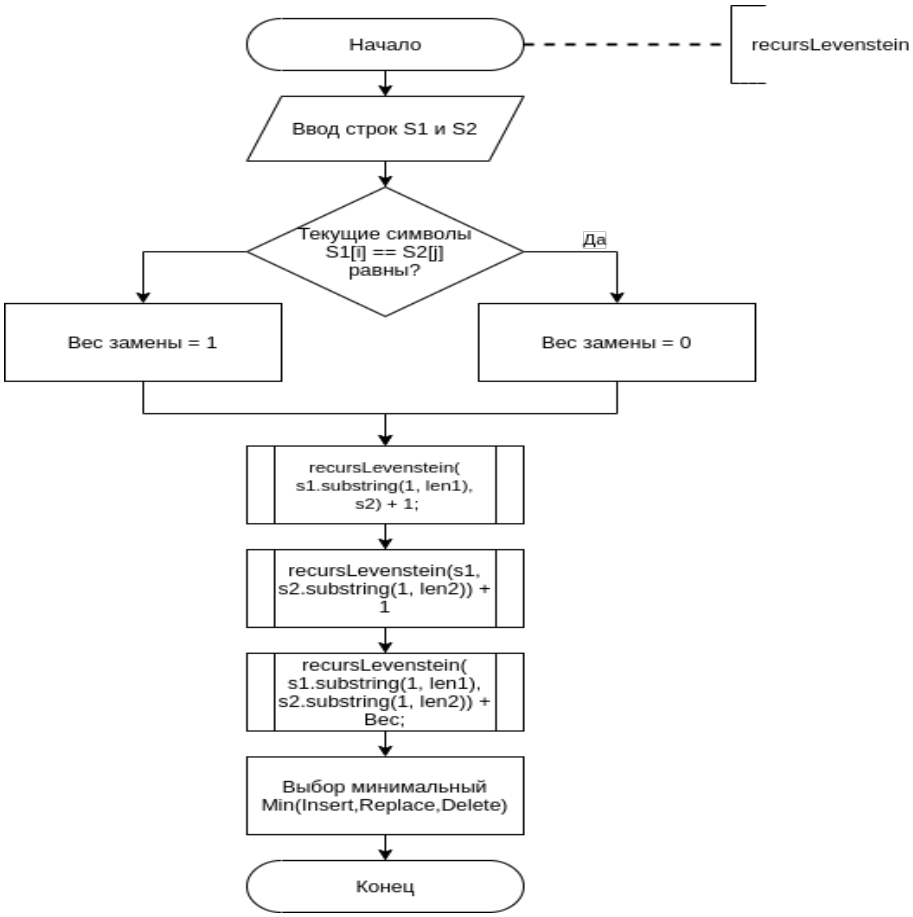

Рис. 3: Схема алгоритма нахождения расстояния Левенштейна рекурсия сделана по реализации кода в приложении.

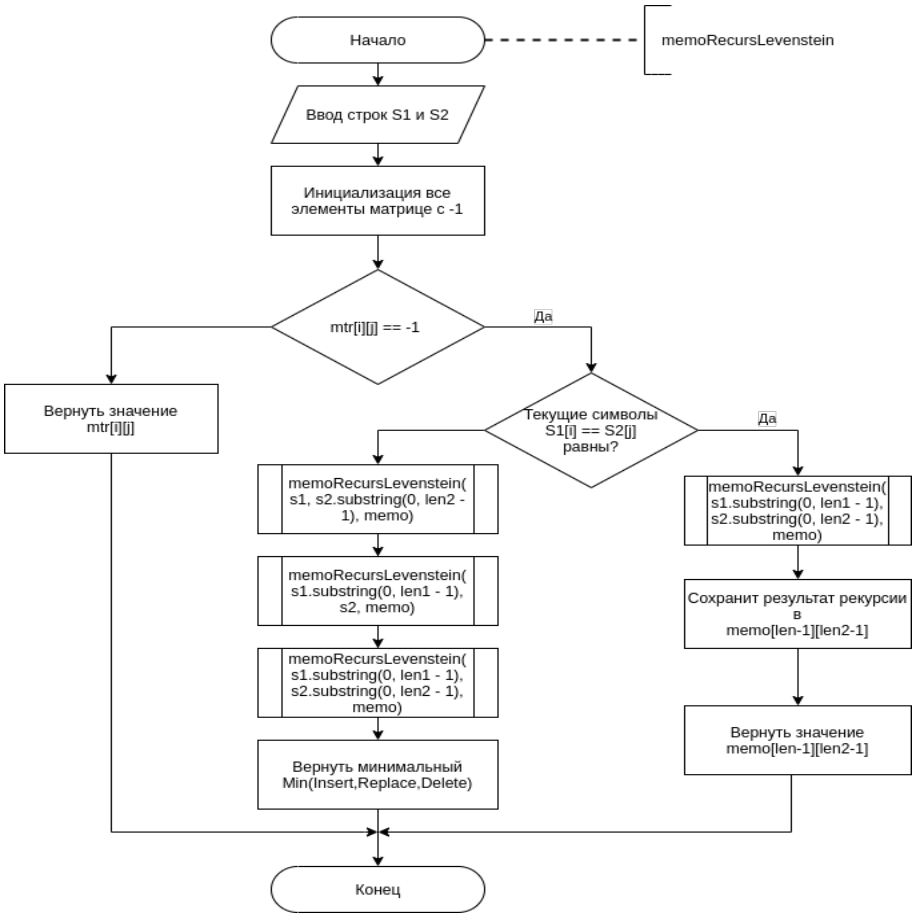

Рис. 4: Схема алгоритма нахождения расстояния Левенштейна рекурсия с меморизации сделана по реализации кода в приложении.

Рекурсивная реализация: начинаем с последнего элемента матрицы D, в котором и содержится искомое расстояние, и рекурсивно находим все недостающие для расчётов элементы.

3 Технологическая часть

3.1 Выбор языка - Java

Java [1] — строго типизированный объектно-ориентированный язык программирования, разработанный компанией Sun Microsystems (в последующем приобретённой компанией Oracle). Разработка ведётся сообществом, организованным через Java Community Process, язык и основные реализующие его технологии распространяются по лицензии GPL. Права на торговую марку принадлежат корпорации Oracle.

Приложения Java обычно транслируются в специальный байт-код, поэтому они могут работать на любой компьютерной архитектуре с помощью виртуальной Java-машины. Дата официального выпуска — 23 мая 1995 года. На 2019 год Java — один из самых популярных языков программирования.

Программы на Java транслируются в байт-код Java, выполняемый виртуальной машиной Java (JVM) — программой, обрабатывающей байтовый код и передающей инструкции оборудованию как интерпретатор.

3.2 Методы замера времени в программе

3.2.1 Время

На листингах 1 – 2 показаны методы измерения времени в джаве.

Листинг 1: Пример System.currentTimeMillis()

```
class Time {
    public static long getCurrentTime() {
        return System.currentTimeMillis();
    }
```

Единственным недостатком этого подхода является то, что "реальное"время doSomething(), которое требуется выполнить, может сильно различаться в зависимости от того, какие другие программы работают в системе и какова его нагрузка. Это делает измерение производительности несколько неточным.

Листинг 2: Пример ThreadMXBean.

```
public class CPUTime {
    public static long getCPUTime() {
        ThreadMXBean bean = ManagementFactory.getThreadMXBean();
        return bean.isCurrentThreadCpuTimeSupported() ?
                bean.getCurrentThreadCpuTime() : 0L;
    }
}
```

Более точный способ отслеживания времени, затрачиваемого на выполнение кода, при условии, что код является однопоточным, - это смотреть на время процессора, потребляемое потоком во время вызова. Это можно сделать с помощью классов ‘JMX‘; в частности, с ‘ThreadMXBean‘. Нужно получить экземпляр ‘ThreadMXBean‘ из ‘java.lang.management.ManagementFactory‘, и если платформа поддерживает его (большинство из них), использование ‘getCurrentThreadCpuTime‘ вместо ‘System.currentTimeMillis‘ выполнить аналогичный тест. ‘getCurrentThreadCpuTime‘ сообщает время в наносекундах, а не миллисекундах.

3.2.2 Улучшение точности замеров времени

Чтобы получить более плавные результаты каждый тест запускается несколько раз.

Java компилируемый и интерпретируемый язык. Изначально javac компилирует код в byte code, а после этого он запускается в JVM. Для улучшения производительности JIT в JVM во время интерпретации компилирует код в нативный для системы. Еще одна особенност JIT, является в том, что для того, чтобы сработали много из оптимизаций нужно выполниться код несколько раз. Для улучшения точности можно выключить оптимизации JVM и JIT или не считать время первых выполнений кода. Для того, чтобы получить время, за которое будет выполняться алгоритм в реальных условиях нужно использовать второй способ.

4 Экспериментальная часть

4.1 Листинг кода

На листингах 3 - 6 показаны реализации алгоритмов для нахождения редакционного растояния.

Листинг 3: Алгоритм для нахождения расстояние Левенштейна итеративная реализация

```
class Alg{
    public static int levenstein(String s1, String s2) {
        int len1 = s1.length();
        int len2 = s2.length();

        if (len1 == 0)
            return len2;
        if (len2 == 0)
            return len1;

        int[][] mtr = new int[len1 + 1][len2 + 1];
        for (int i = 0;  i < len1 + 1; i++) {
            mtr[i][0] = i;
        }

        for (int i = 0; i < len2 + 1; i++) {
            mtr[0][i] = i;
        }


        int turn = 0;
        for (int i = 1; i < len1 + 1; i++) {
            for (int j = 1; j < len2 + 1; j++) {
                turn = s1.charAt(i - 1) == s2.charAt(j - 1) ? 0 : 1;
                mtr[i][j] = Math.min(
                                Math.min(
                                        mtr[i - 1][j] + 1,
                                        mtr[i][j - 1] + 1
                                        ),
                                mtr[i - 1][j - 1] + turn);

            }
        }
        return mtr[len1][len2];
    }
}
```

Листинг 4: Алгоритм для нахождения расстояние Левенштейна рекурсивная реализация

```
class Alg {
    public static int recursLevenstein(String s1, String s2) {
        int len1 = s1.length();
        int len2 = s2.length();

        if (len1 == 0)
            return len2;

        if (len2 == 0)
            return len1;

        int cost = s1.charAt(0) == s2.charAt(0) ? 0 : 1;

        int insert = recursLevenstein(s1.substring(1, len1), s2) + 1;
        int delete = recursLevenstein(s1, s2.substring(1, len2)) + 1;
        int replace = recursLevenstein(s1.substring(1, len1),
                            s2.substring(1, len2)) + cost;

        return Math.min(delete, Math.min(insert, replace));
    }
}
```

Листинг 5: Алгоритм для нахождения расстояние Левенштейна рекурсивная реализация с заполнением матрицы

```
class Alg {
   public static int memoRecursLevenstein(String s1, String s2, int memo[][]) {
        int len1 = s1.length();
        int len2 = s2.length();

        if (memo == null) {
            memo = new int[len1][len2];
            matirxInit(memo, len2, -1);
        }

        if (len1 == 0)
            return len2;

        if (len2 == 0)
            return len1;

        if (memo[len1 - 1][len2 - 1] != -1) {
            return memo[len1 - 1][len2 - 1];
        }

        if (s1.charAt(len1 - 1) == s2.charAt(len2 - 1))
            return memo[len1 - 1][len2 - 1] = memoRecursLevenstein(
                        s1.substring(0, len1 - 1),
                        s2.substring(0, len2 - 1),
                                            memo);
        return memo[len1 - 1][len2 - 1] = 1 +
                min(
                        memoRecursLevenstein(
                                s1,
                                s2.substring(0, len2 - 1),
                                memo), //insert,
                        memoRecursLevenstein
                                s1.substring(0, len1 - 1),
                                s2,
                                memo),//Remove
                        memoRecursLevenstein(
                                s1.substring(0, len1 - 1),
                                s2.substring(0, len2 - 1),
                                memo) // Replace
                );
    }
}
```

Листинг 6: Алгоритм для нахождения расстояние Дамерау - Левенштейна итеративная реализация

```
class Alg {
    public static int damerauLevenstein(String s1, String s2) {
        int len1 = s1.length();
        int len2 = s2.length();

        if (len1 == 0)
            return len2;

        if (len2 == 0)
            return len1;

        int[][] mtr = new int[len1 + 1][len2 + 1];
        for (int i = 0;  i < len1 + 1; i++) {
            mtr[i][0] = i;
        }

        for (int i = 0; i < len2 + 1; i++) {
            mtr[0][i] = i;
        }

        int turn = 0;
        for (int i = 1; i < len1 + 1; i++) {
            for (int j = 1; j < len2 + 1; j++) {
                turn = s1.charAt(i - 1) == s2.charAt(j - 1) ? 0 : 1;

                int insert = mtr[i - 1][j] + 1;
                int delete = mtr[i][j - 1] + 1;
                int replace = mtr[i - 1][j - 1] + turn;

                int min = Math.min(delete, Math.min(insert, replace));
                if (i > 1 && j > 1 &&
                        s1.charAt(i - 1) == s2.charAt(j - 2) &&
                        s1.charAt(i - 2) == s2.charAt(j - 1) &&
                        turn == 1) {
                    min = Math.min(min, mtr[i - 2][j - 2] + 1);
                }
                mtr[i][j] = min;
            }

        }

        return mtr[len1][len2];
    }
}
```

4.2 Примеры работы

На рисунке 5 - 6 приведены примеры работы программы.
Обозначения:

‘DL‘ - расстояние Дамерау-Левенштейна;

‘L‘ - расстояние Левенштейна;

‘RL‘ - рекурсивный алгоритм нахождения расстояния Левенштейна;

Дамерау-Левенштейна	Левенштейна

Enter S1: aaaaa
Enter S2: bbb
 	
  b b b
  0 1 2 3
a 1 1 2 3
a 2 2 2 3
a 3 3 3 3
a 4 4 4 4
a 5 5 5 5
 	
Answer DL: 5
Answer RDL: 5
	Enter S1: aaaaa
	Enter S2: bbb
	
	  b b b
	  0 1 2 3
	a 1 1 2 3
	a 2 2 2 3
	a 3 3 3 3
	a 4 4 4 4
	a 5 5 5 5
	
	Answer L: 5
	Answer RL: 5

Enter S1: acre
Enter S2: car
 	
  c a r
  0 1 2 3
a 1 1 1 2
c 2 1 1 2
r 3 2 2 1
e 4 3 3 2
 	
Answer DL: 2
Answer RDL: 2
	Enter S1: acre
	Enter S2: car
	
	  c a r
	  0 1 2 3
	a 1 1 1 2
	c 2 1 2 2
	r 3 2 2 2
	e 4 3 3 3
	
	Answer L: 3
	Answer RL: 3

Enter S1: ok
Enter S2: ko
 	
  k o
  0 1 2
o 1 1 1
k 2 1 1
 	
Answer DL: 1
Answer RDL: 1
	Enter S1: ok
	Enter S2: ko
	
	  k o
	  0 1 2
	o 1 1 1
	k 2 1 2
	
	Answer L: 2
	Answer RL: 2

Рис. 5: Примеры работы алгоритмов нахождения растоянии Левенптейна и ДамерауЛевенштейна

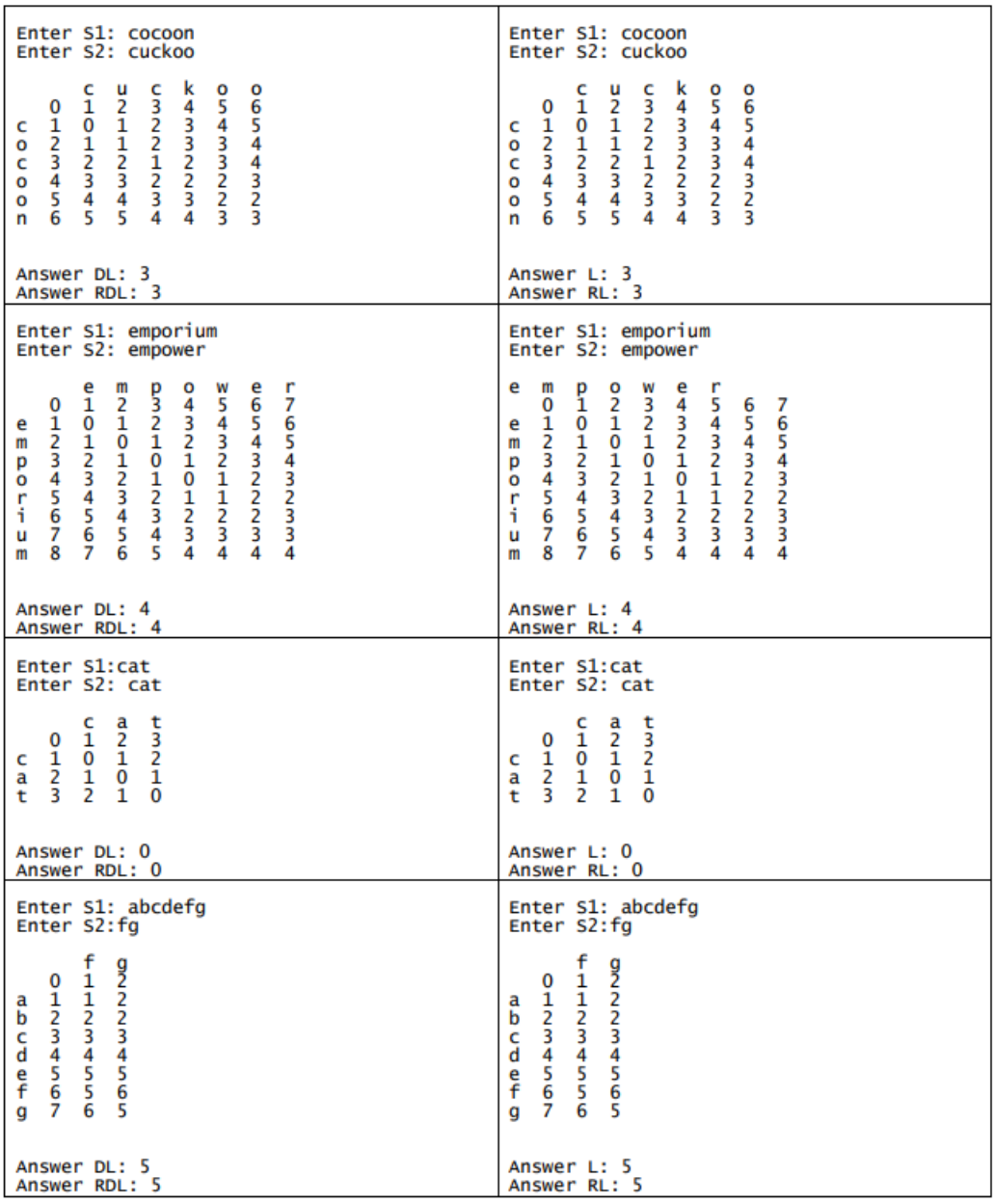

Рис. 6: Примеры работы алгоритмов нахождения растоянии Левенптейна и ДамерауЛевенштейна

4.3 Показательное сравнение временных характеристик

Сравнение быстродействия алгоритмов Левенштейна и Дамерау – Левенштейна в матричной и рекурсивной формой в тиках приведено в таблице 1.

Таблица 1: Временные сравнения алгоритмов

Слова	L	LR	LRM	DL
5	45205	221287	43638	50226
10	65406	281802409	75093	69398
15	70285	480715633	74295	74147
100	749701	–	5300996	1318668
200	1257174	–	13156763	1474800
350	2248876	–	43642112	6531504
500	19321097	–	114866596	31981314

Для алгоритма поиска расстояния Левенштейна рекурсивную реализацию были проведены замеры времени на строках с размером до 15 символов. Характеристики компютера на котором производились замеры времени:

1. операционная система – Ubuntu 19.10;
2. процессор – Intel(R) Core(TM) i5-4210M CPU @ 2.60GHz;
3. видеокарта – Nvidia GeForce 840m;

4.4 Показательное сравнение по памяти

Алгоритмы Левенштейна и Дамерау – Левенштейна не отличаются друг от друга с точки зрения использования памяти, следовательно, достаточно рассмотреть лишь разницу рекурсивной и матричной реализаций этих алгоритмов
Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк, соответственно, максимальный расход памяти:

$$C(S_1) + C(S_2)) \cdot (2 \cdot C(string) + 3 \cdot C(int) \tag{3}$$

где C – оператор вычисления размера, S_1, S_2 – строки, int – целочисленый тип, string – строковой тип
Использование памяти при итеративной реализации теоритически равно:

$$C(S_1 + 1) + C(S_2 + 1)) \cdot C(int) + 10 \cdot C(int) + 2 \cdot C(string) \tag{4}$$

Заключение

В ходе работы был сделан вывод, что рекурсивная реализация алгоритма Левенштейна и Дамерау – Левенштейна выполняется за приемлемое время только в случаях, когда размер одной из строк крайне мал, а так же что итеративные реализации алгоритмов поиска расстояний Демерау – Левенштейна и Левенштейна имеют схожую конструкцию, но алгоритм поиска расстояния Демерау – Левенштейн из-за более сложной внутренней логики в среднем работает медленнее. Но оба алгоритмы являются на много эффективнее по времени от их рекурсивных реализаций.

Была достигнута цель и решены следующие задачи:

1) были расмотрены алгоритмы поиска расстояние Левенштейна и Демерау - Левенштейна для нахождения редакционного расстояния между строками;

2) были реализованы данные алгоритмы на языков программирования Java;

3) был проведен сравнительный анализ алгоритмов по затраченному времени и памяти;

4) были приведены примеры работы всех указанных алгоритмов;

Список литературы

1. Oracle: Java. [ЭЛ. РЕСУРС]
Режим доступа: https://www.oracle.com/java/
(дата обращения: 09.09.2020).

2. Карахтанов, Д. С. Программная реализация алгоритма Левенштейна для устранения опечаток в записях баз данных. [ЭЛ.РЕСУРС]
Режим доступа: https://moluch.ru/archive/19/1966/
(дата обращения: 09.09.2020).

3. D a m e r a u F. A. Technique for Computer Detection and Correction of Spelling Errors
Communications of the ACM. 1964. Vol. 7. No. 3. P. 171–176.
(дата обращения: 09.09.2020).