МГТУ им. Баумана

Лабораторная работа №1

По курсу: "Анализ алгоритмов"

Расстояние Левенштейна

Работу выполнил: Гаврилов Дмитрий, ИУ7-56Б

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

Москва, 2019

Оглавление

Введение

Расстояние Левенштейна - минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

Расстояние Левенштейна применяется в теории информации и компьютерной лингвистике для:

- исправления ошибок в слове
- сравнения текстовых файлов утилитой diff
- в биоинформатике для сравнения генов, хромосом и белков

Целью данной лабораторной работы является изучение метода динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна.

Задачами данной лабораторной являются:

1. изучение алгоритмов Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;
2. применение метода динамического программирования для матричной реализации указанных алгоритмов;
3. получение практических навыков реализации указанных алгоритмов: двух алгоритмов в матричной версии и одного из алгоритмов в рекурсивной версии;
4. сравнительный анализ линейной и рекурсивной реализаций выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);

5. экспериментальное подтверждение различий во временно́й эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк;

6. описание и обоснование полученных результатов в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

1 Аналитическая часть

Задача по нахождению расстояния Левенштейна заключается в поиске минимального количества операций вставки/удаления/замены для превращения одной строки в другую.

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

Действия обозначаются так:

1. D (англ. delete) — удалить,
2. I (англ. insert) — вставить,
3. R (replace) — заменить,
4. M(match) - совпадение.

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

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

где $m(a, b)$ равна нулю, если $a = b$ и единице в противном случае; $min\{\, a, b, c\}$ возвращает наименьший из аргументов.

Расстояние Дамерау-Левенштейна вычисляется по следующей рекуррентной формуле:

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

1.0.1 Вывод

В данном разделе были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна, который является модификаций первого, учитывающего возможность перестановки соседних символов.

2 Конструкторская часть

Требования к вводу:

1. На вход подаются две строки
2. uppercase и lowercase буквы считаются разными

Требования к программе:

1. Две пустые строки - корректный ввод, программа не должна аварийно завершаться

2.1 Схемы алгоритмов

В данной части будут рассмотрены схемы алгоритмов.

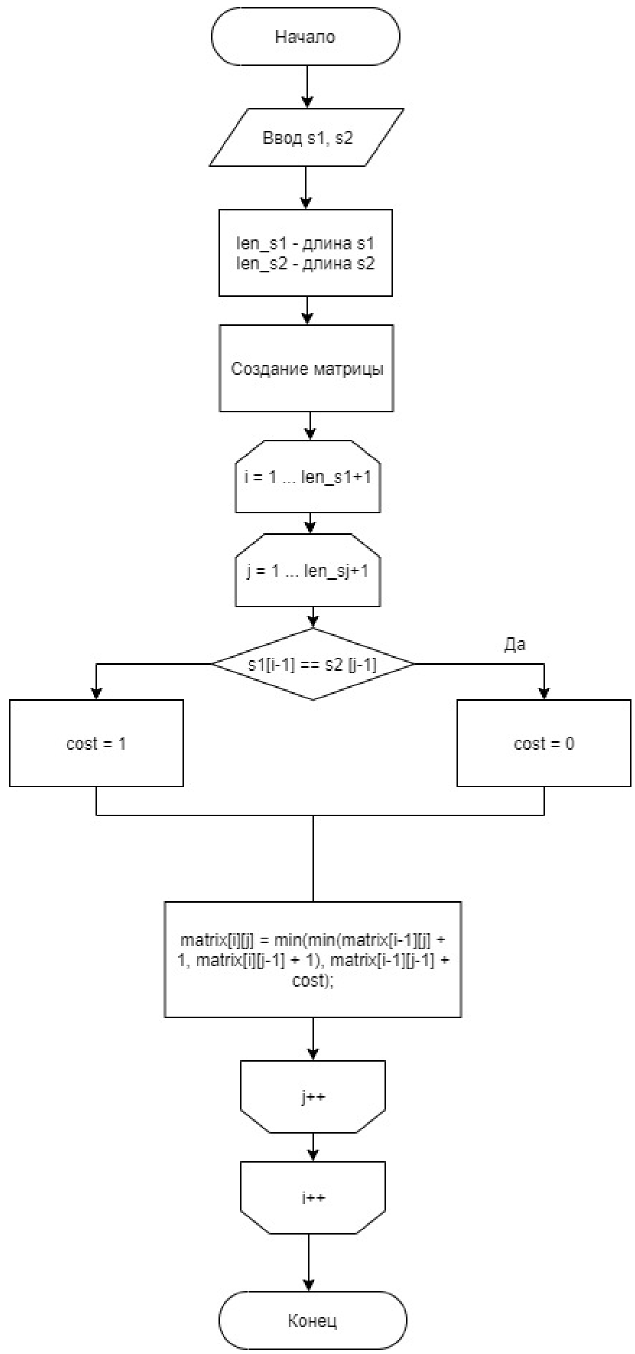


Рис. 2.1: Схема матричного алгоритма нахождения расстояния Левенштейна

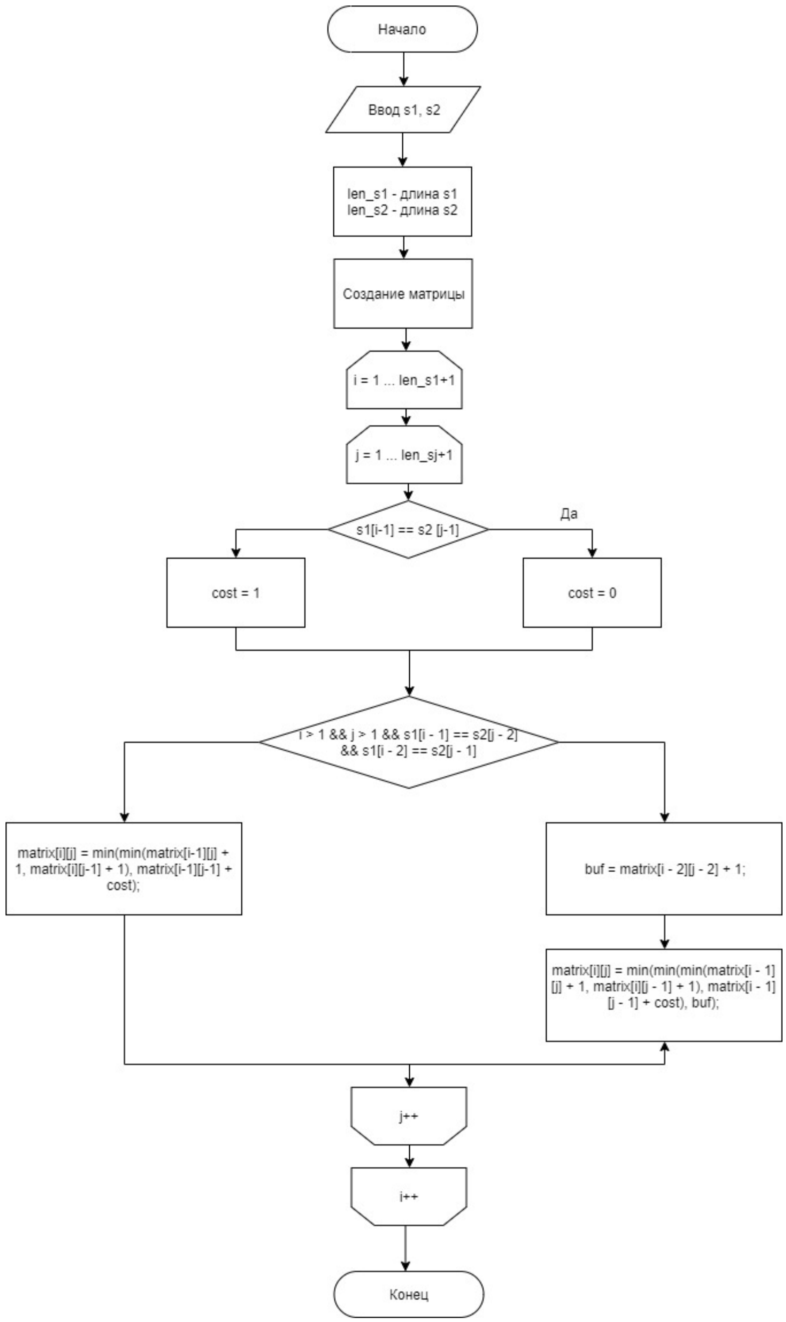


Рис. 2.2: Схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна

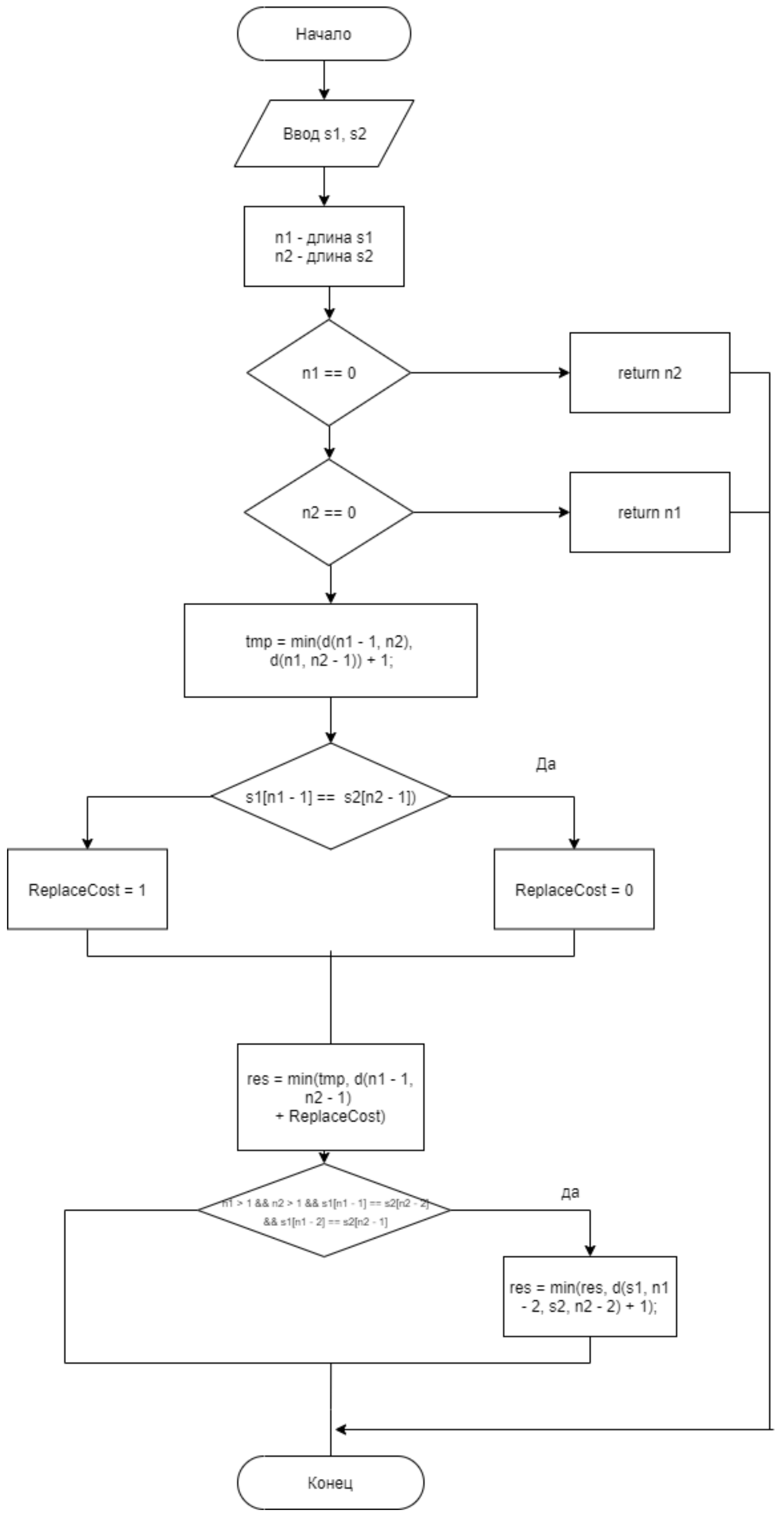


Рис. 2.3: Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

3 Технологическая часть

3.1 Выбор ЯП

Для реализации программ я выбрал язык программирования Java, так имею большой опыт работы с ним. Среда разработки - Intellij IDEA.

3.2 Реализация алгоритма

Листинг 3.1: Класс нахождения расстояния Левенштейна матрично

```
public class LowensteinDistance extends DistanceBase {
    protected int[][] matrix;

    public LowensteinDistance(String firstWord, String
       secondWord) {
        super(firstWord, secondWord);
    }

    @Override
    protected int calculateDistance() {
        matrix = new int[firstWord.length() + 1][secondWord
            .length() + 1];
        fillMatrix();
        findDistanceInMatrix();
        return getResultFromMatrix();
    }

    protected void getNextElementInMatrix(int i, int j) {
        int cost = calculateCost(i, j);

```

```java
        matrix[i][j] = Math.min(Math.min(matrix[i - 1][j] + 1, matrix[i][j - 1] + 1), matrix[i - 1][j - 1] + cost);
    }

    protected int calculateCost(int i, int j) {
        return (firstWord.charAt(i - 1) == (secondWord.charAt(j - 1))) ? 0 : 1;
    }

    private void findDistanceInMatrix() {
        for (int i = 1; i <= firstWord.length(); i++) {
            for (int j = 1; j <= secondWord.length(); j++) {
                getNextElementInMatrix(i, j);
            }
        }
    }

    private void fillMatrix() {
        fillMatrixFirstColumn();
        fillMatrixFirstRow();
    }

    private void fillMatrixFirstColumn() {
        for (int i = 0; i < firstWord.length() + 1; i++) {
            matrix[i][0] = i;
        }
    }

    private void fillMatrixFirstRow() {
        for (int i = 0; i < secondWord.length() + 1; i++) {
            matrix[0][i] = i;
        }
    }

    private int getResultFromMatrix() {
        return matrix[firstWord.length()][secondWord.length()];
    }
```

```
    @Override
    public String toString() {

        StringBuilder distanceMatrix = new StringBuilder("\n\n");
        for (int i = 0; i < matrix.length; i++)
        {
            for (int j = 0; j < matrix.length; j++)
                distanceMatrix.append(matrix[i][j] + "\t");
            distanceMatrix.append("\n");
        }
        distanceMatrix.append("\n\n");

        return distanceMatrix.toString();
    }
}
```

Листинг 3.2: Класс нахождения расстояния Дамерау-Левенштейна матрично

```
public class LowensteinDamerauDistance extends LowensteinDistance {
    public LowensteinDamerauDistance(String firstWord, String secondWord) {
        super(firstWord, secondWord);
    }

    @Override
    protected void getNextElementInMatrix(int i, int j) {
        super.getNextElementInMatrix(i, j);

        if (adjacentLetterEqual(i, j))
            matrix[i][j] = Math.min(matrix[i][j], matrix[i - 2][j - 2] + calculateCost(i, j));
    }

    private boolean adjacentLetterEqual(int i, int j) {
        if (i > 1 && j > 1 &&
                firstWord.charAt(i - 1) == secondWord.charAt(j - 2) &&
```

```
                    firstWord.charAt(i - 2) == secondWord.
                        charAt(j - 1)) {
                return true;
            }
            return false;
        }
}
```

Листинг 3.3: Класс нахождения расстояния Дамерау-Левенштейна рекурсивно

```
public class LowensteinRecursiveDistance extends
    DistanceBase {
    public LowensteinRecursiveDistance(String firstWord,
        String secondWord) {
        super(firstWord, secondWord);
    }

    @Override
    protected int calculateDistance() {
        return calculateRecursive(firstWord.length(),
            secondWord.length());
    }

    private int calculateRecursive(int firstWordLength, int
        secondWordLength) {
        int cost;

        if (firstWordLength == 0)
            return secondWordLength;
        if (secondWordLength == 0)
            return firstWordLength;

        if (firstWord.charAt(firstWordLength - 1) ==
            secondWord.charAt(secondWordLength - 1))
            cost = 0;
        else
            cost = 1;


        return Collections.min(Arrays.asList(
```

```
                    calculateRecursive(firstWordLength - 1,
                        secondWordLength) + 1,
                    calculateRecursive(firstWordLength,
                        secondWordLength - 1) + 1,
                    calculateRecursive(firstWordLength - 1,
                        secondWordLength - 1) + cost
            ));
        }
}
```

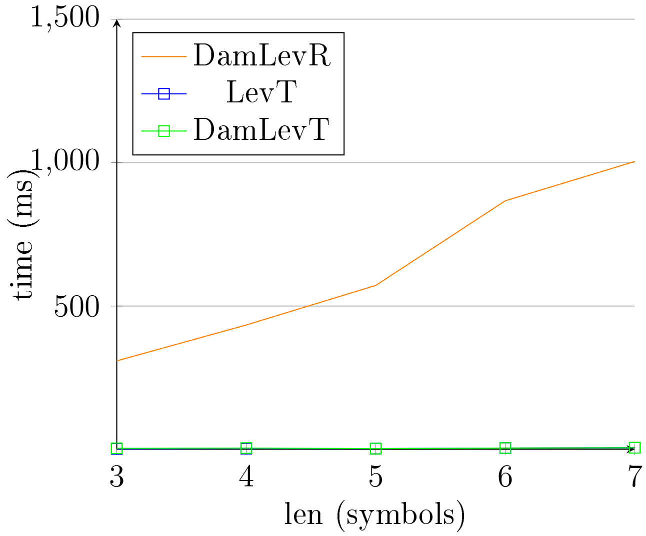
4 Исследовательская часть

4.1 Сравнительный анализ на основе замеров времени работы алгоритмов

Был проведен замер времени работы каждого из алгоритмов.

Таблица 4.1: Время работы алгоритмов (в милисекундах)

len	Lev(M)	DamLev(M)	DamLev(R)
3	2	4	309
4	3	5	434
5	3	3	572
6	5	5	867
7	6	7	1004



Наиболее эффективными по времени являются матричные реализации алгоритмов, уже при длине строк в 7 символов выигрышность становится более чем в 1,000 раз. Это обусловлено большим количеством повторных рассчетов рекурсивных алгоритмов. Время работы алгоритма, использующего матрицу, намного меньше благодаря тому, что в нем требуется только (m + 1)*(n + 1) операций заполнения ячейки матрицы. Также установлено, что алгоритм ДамерауЛевенштейна работает немного дольше алгоритма Левенштейна, т.к. в нем добавлены дополнительные проверки, однако алгоритмы сравнимы по временной эффективности.

4.2 Сравнительный анализ на основе замеров потребляемой памяти алгоритмов

Для проведения анализа замерим потребляемую память у разных реализаций алгоритма. Все измерения представлены в байтах

Таким образом, рекурсивный алгоритм занимает примерно одинаковое кол-во памяти при маленькой длине строк, и значительно выигрывает при строках большего размера.

Таблица 4.2: Потребляемая память структурами данных в алгоритме нахождения расстояния Левенштейна

Структура данных	Длина 4 символа	Длина 1000 символов
Матрица	480	8064096
Две вспомогательные переменные (int)	56	56
Два счетчика (int)	56	56
Передача параметров	106	2098
Сумма данных	698	8066306

Таблица 4.3: Потребляемая память структурами данных в алгоритме нахождения расстояния Дамерау-Левенштейна

Структура данных	Длина 4 символа	Длина 1000 символов
Матрица	480	8064096
Три вспомогательные переменные (int)	84	84
Два счетчика (int)	56	56
Передача параметров	106	2098
Сумма данных	726	8066334

Таблица 4.4: Потребляемая память структурами данных в рекурсивном алгоритме нахождения расстояния Дамерау-Левенштейна

Структура данных	Длина 4 символа	Длина 1000 символов
Пять переменных для подсчета IDTR	$140 * 4 = 560$	$140 * 1000 = 140000$
Передача параметров	$106 * 8 = 848$	$2098 * 2000 = 4196000$
Сумма данных	1408	4336000

4.3 Тестовые данные

Проведем тестирование программы. В столбцах "Ожидаемый результат"и "Полученный результат"3 числа соответсвуют матричному алгоритму нахождению расстоянию Левенштейна, рекурсивному алгоритму расстояния Дамерау-Левенштейна, матричному алгоритму нахождения расстояние Дамерау-Левенштейна.

Таблица 4.5: Таблица тестовых данных

№	Первое слово	Второе слово	Ожидаемый результат	Полученный результат
1			0 0 0	0 0 0
2	kot	skat	2 2 2	2 2 2
3	kate	ktae	2 1 1	2 1 1
4	abacaba	aabcaab	4 2 2	4 2 2
5	sobaka	sboku	3 3 3	3 3 3
6	qwerty	queue	4 4 4	4 4 4
7	apple	aplpe	2 1 1	2 1 1
8		cat	3 3 3	3 3 3
9	parallels		9 9 9	9 9 9
10	bmstu	utsmb	4 4 4	4 4 4

Заключение

Был изучен метод динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна. Также изучены алгоритмы Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками, получены практические навыки раелизации указанных алгоритмов в матричной и рекурсивных версиях.

Экспериментально было подтверждено различие во временной эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработаного программного обеспечения на материале замеров времени выполнения реализации на варьирующихся длинах строк.

В результате исследований я пришел к выводу, что матричная реализация данных алгоритмов заметно выигрывает по времени при росте длины строк, следовательно более применима в реальных проектах.