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» (ИУ7)**

НАПРАВЛЕНИЕ ПОДГОТОВКИ **09.03.01 Информатика и вычислительная техника**

О Т Ч Е Т

по лабораторной работе № 1

Название: Расстояние Левенштейна и Дамерау-Левенштейна

Дисциплина: Анализ алгоритмов

Студент	ИУ7-52Б (Группа)	(Подпись, дата)	В.А. Иванов (И.О. Фамилия)
Преподаватель		(Подпись, дата)	(И.О. Фамилия)

Москва, 2020

Оглавление

Введение

Расстояние Левенштейна – минимальное количество редакционных операций, которые необходимы для превращения одной строки в другую. Существуют следующие редакционные операции:

- вставка символа;
- удаление символа;
- замена символа;

Расстояние Дамерау-Левенштейна также учитывает и операцию транспозиции – перестановки двух соседних символов местами.

Данные расстояния имеют большое количество применений. Они используются для автокоррекции при выполнении поисковых запросов и печати на клавиатуре, а также в биоинформатике для сравнения генов, представленных в строковом формате.

1. Аналитическая часть

Целью лабораторной работы является реализация и сравнение алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.

Выделены следующие задачи лабораторной работы:

- математическое описание расстояний Левенштейна и Дамерау-Левенштейна;
- описание и реализация алгоритмов поиска расстояний;
- проведение замеров процессорного времени работы алгоритмов при различных размерах строк;
- оценка наибольшей используемой каждым алгоритмом памяти;
- проведение сравнительного анализа алгоритмов на основании экспериментов;

Задача по поиску расстояний заключается в нахождении такой последовательности операций, применение которых даст минимальный суммарный штраф. Штрафы операций:

- вставка (I) – 1;
- замена (R) – 1;
- удаление (D) – 1;
- совпадение (M) – 0;
- транспозиция (T) – 1;

Для решения данной проблемы используется рекуррентная формула вычисления расстояний. Пусть D(s1[1..i], s2[1..j]) – расстояние Левенштейна для подстроки s1 длиной i и s2 длиной j. Формула вычисления D:

$$\begin{cases} j, \quad \text{если } i = 0 \\ i, \quad \text{если } j = 0 \\ \min(D(s1[1..i], \quad s2[1..j-1]) + 1, \\ D(s1[1..i-1], \quad s2[1..j]) + 1, \\ D(s1[1..i-1], \quad s2[1..j-1]) + \begin{cases} 0, \text{если } s1[i] == s2[j] \\ 1, \text{иначе} \end{cases}) \end{cases}$$

Аналогично рекуррентно представляется формула расстояния Дамерау-Левенштейна:

$$
\begin{cases}
j, \quad \text{если } i = 0 \\
i, \quad \text{если } j = 0 \\
\min\big(D(s1[1..i], \quad s2[1..j-1]) + 1, \\
D(s1[1..i-1], \quad s2[1..j]) + 1, \\
D(s1[1..i-1], s2[1..j-1]) + \begin{cases} 0, \text{если } s1[i] = s2[j] \\ 1, \text{иначе} \end{cases}, \\
\begin{cases}
D(s1[1..i-2], s2[1..j-2]) + 1\big), \quad \text{если} \begin{cases} i > 1, j > 1 \\ s1[i] = s2[j-1] \\ s1[i-1] = s2[j] \end{cases} \\
+\infty, \quad \text{иначе}
\end{cases}
\end{cases}
$$

2. Конструкторская часть

Рассмотрим алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна для строк s1 и s2 с длинами в n1 и n2 соответственно.

2.1. Расстояние Левенштейна, матричный метод

Алгоритм матричного поиска расстояния Левенштейна основывается на вышеописанной рекуррентной формуле. Создаётся целочисленная матрица размерами (n1+1)x(n2+1). В каждой клетке [i][j] этой матрицы будет записано значение D(s1[1..i-1], s2[1..j-1]). В случае, когда i=1 или j=1 вместо строк s1 и s2 соответственно будут выступать пустые строки. Искомым расстоянием Левенштейна будет значение ячейки [n1+1][n2+1].

Нахождение расстояний алгоритм начинает с заполнения первого столбца и первой строки, так как они являются базой для рекуррентной формулы. После этого производится построчное заполнение остальной части матрицы.

2.2. Расстояние Дамерау-Левенштейна, матричный метод

Алгоритм является модификацией вышеописанного способа нахождения расстояния Левенштейна. Дополнительно для ячейки [i][j] (i > 2, j > 2) рассматривается вариант перехода из клетки [i-2][j-2], при условии, что s1[i] = s2[j-1] и s1[i-1] = s2[j]. Искомым расстоянием Дамерау-Левенштейна также является значение ячейки [n1+1][n2+1].

2.3. Расстояние Левенштейна, рекурсивный метод

Данный алгоритм использует только рекурсивную формулу нахождения D(s1[1..i], s2[1..j]). Для этого используется рекурсивная функция, принимающая в себя строки s1, s2 и длины подстрок i, j. Функция вызывает функции для тех же строк, и длин: (i-1, j-1), (i-1, j) и (i, j-1), после чего возвращает минимальный из них.

2.4. Расстояние Левенштейна, рекурсивный метод с заполнением матрицы

В данном случае, в качестве основы используется алгоритм Дейкстры поиска расстояний до вершин в графе. Создаётся матрица размерами (n1+1)x(n2+1), все ячейки которой изначально заполнены значением +∞. В каждой клетке [i][j] этой матрицы будет записано значение D(s1[1..i-1], s2[1..j-1]).

Рекурсивная функция получает матрицу, индексы i, j положения в ней и две строки. Алгоритм начинает свою работу с ячейки [1][1], которая заполняется значением 0. Из положения [i][j] рассматривается переход в соседние ячейки [i+1][j+1], [i+1][j], [i][j+1]. В случае, если соседняя ячейка расположена в пределах матрицы, и расстояние R при переходе из данной ячейки меньше ныне хранимого в ней, то значение соседней ячейки меняется на R, после чего функция запускается уже для соседней ячейки. После завершения работы всех функций, расстояние Левенштейна расположено в ячейке [n1+1][n2+1].

2.5. Требования к ПО

Для полноценной проверки и оценки алгоритмов необходимо выполнить следующее.

1. Обеспечить возможность консольного ввода двух строк и выбора алгоритма для поиска расстояния. Программа должна вывести вычисленное редакционное расстояние, а также вывести матрицу поиска, в случае использования её в выбранном алгоритме.
2. Реализовать функцию замера процессорного времени, затраченного функцией. Для этого также создать возможность ввода длины строк, на которых будет выполнен замер.

3. Технологическая часть

3.1. Выбор языка программирования

В качестве языка программирования был выбран Python 3, так как имеется опыт работы с ним, и с библиотеками, позволяющими провести исследование и тестирование программы.

3.2. Листинг кода

Реализация алгоритмов поиска расстояний представлена на листингах 3.1-3.4.

Листинг 3.1. Функция нахождения расстояния Левенштейна матричным методом.

```
def lev_matrix(s1, s2, is_print=False):
    matr = [[0] * (len(s1)+1) for i in range(len(s2)+1)]

    for j in range(len(s1)+1):
        matr[0][j] = j
    for i in range(len(s2)+1):
        matr[i][0] = i

    for i in range(1, len(s2)+1):
        for j in range(1, len(s1)+1):
            add = 0 if s1[j-1] == s2[i-1] else 1
            matr[i][j] = min(matr[i-1][j]+1, matr[i][j-1]+1, matr[i-1][j-1]+add)

    if is_print:
        print("Расстояние:", matr[i][j])
        print_matrix(matr)
    return matr[i][j]
```

Листинг 3.2. Функции нахождения расстояния Левенштейна рекурсивным методом.

```
def _lev_rec(s1, s2, len1, len2):
    if len1 == 0: return len2
    elif len2 == 0: return len1
    else:
        return min(_lev_rec(s1, s2, len1, len2-1) + 1,
                   _lev_rec(s1, s2, len1-1, len2) + 1,
                   _lev_rec(s1, s2, len1-1, len2-1) +
```

```
            (0 if s1[len1-1] == s2[len2-1] else 1))

def lev_recursion(s1, s2, is_print=False):
    res = _lev_rec(s1, s2, len(s1), len(s2))
    if is_print:
        print("Расстояние:", res)
    return res
```

Листинг 3.3. Функции нахождения расстояния Левенштейна рекурсивным методом с заполнением матрицы.

```
def _lev_mr(matr, i, j, s1, s2):
    if i+1 < len(matr) and j+1 < len(matr[0]):
        add = 0 if s1[j] == s2[i] else 1
        if matr[i+1][j+1] > matr[i][j] + add:
            matr[i+1][j+1] = matr[i][j] + add
            _lev_mr(matr, i+1, j+1, s1, s2)
    if j+1 < len(matr[0]) and (matr[i][j+1] > matr[i][j] + 1):
        matr[i][j+1] = matr[i][j] + 1
        _lev_mr(matr, i, j+1, s1, s2)
    if i+1 < len(matr) and (matr[i+1][j] > matr[i][j] + 1):
        matr[i+1][j] = matr[i][j] + 1
        _lev_mr(matr, i+1, j, s1, s2)
def lev_matrix_recursion(s1, s2, is_print=False):
    max_len = max(len(s1), len(s2)) + 1
    matr = [[max_len] * (len(s1) + 1) for i in range(len(s2) + 1)]
    matr[0][0] = 0
    _lev_mr(matr, 0, 0, s1, s2)

    if is_print:
        print("Расстояние:", matr[-1][-1])
        print_matrix(matr)
    return matr[-1][-1]
```

Листинг 3.4. Функции нахождения расстояния Дамерау-Левенштейна матричным методом.

```
def dem_lev_matrix(s1, s2, is_print=False):
    matr = [[0] * (len(s1) + 1) for i in range(len(s2) + 1)]
    for j in range(len(s1)+1):
        matr[0][j] = j
    for i in range(len(s2)+1):
        matr[i][0] = i
```

```
    for i in range(1, len(s2) + 1):
        addM = 0 if s1[0] == s2[i-1] else 1
        matr[i][1] = min(matr[i-1][1] + 1, matr[i][0] + 1,
                         matr[i-1][0] + addM)
    for j in range(2, len(s1) + 1):
        addM = 0 if s1[j-1] == s2[0] else 1
        matr[1][j] = min(matr[0][j] + 1, matr[1][j-1] + 1,
                         matr[0][j-1] + addM)

    for i in range(2, len(s2)+1):
        for j in range(2, len(s1)+1):
            addM = 0 if s1[j-1] == s2[i-1] else 1
            addT = 1 if (s1[j-2] == s2[i-1] and s1[j-1] == s2[i-2]) else 2
            matr[i][j] = min(matr[i-1][j]+1, matr[i][j-1]+1,
                             matr[i-1][j-1]+addM, matr[i-2][j-2]+addT)

    if is_print:
        print("Расстояние:", matr[i][j])
        print_matrix(matr)
    return matr[i][j]
```

3.3. Результаты тестирования

Для тестирования написанных функций была использована библиотека unittest. Тестирование функций проводилось за счёт сравнения результата, возвращённого функцией и ожидаемого расстояния для разных наборов строк. Состав тестов приведён в листинге 3.5.

Листинг 3.5. Модульные тесты

```
import unittest
import main

# Общий набор тестов для всех алгоритмов
class GeneralTest(unittest.TestCase):
    # Данный класс являтся абстрактным, поэтому для него тесты
пропускаются
    @unittest.skip("Skip GeneralTest")
    def setUp(self):
        self.function = None

    # Проверка пустыми строками
    def test_empty(self):
```

```
        self.assertEqual(self.function("", ""), 0)
        self.assertEqual(self.function("a", ""), 1)
        self.assertEqual(self.function("", "b"), 1)

    # Проверка нахождения совпадений
    def test_match(self):
        self.assertEqual(self.function("abc", "abc"), 0)
        self.assertEqual(self.function("a", "a"), 0)
        self.assertEqual(self.function("A", "a"), 1)

    # Прочие общие тесты
    def test_other(self):
        self.assertEqual(self.function("q", "w"), 1)
        self.assertEqual(self.function("aq", "aw"), 1)
        self.assertEqual(self.function("a", "aw"), 1)
        self.assertEqual(self.function("aw", "a"), 1)


# Набор тестов для алгоритмов поиска расстояния Левенштейна
class LevTest(GeneralTest):
    def test_lev(self):
        self.assertEqual(self.function("stolb", "telo"), 3)
        self.assertEqual(self.function("kult_tela", "tela_kult"), 6)
        self.assertEqual(self.function("развлечение", "увлечение"), 3)


# Набор тестов для алгоритма поиска расстояния Дамерау-Левенштейна
class DemLevMatrixTest(GeneralTest):
    def setUp(self):
        self.function = main.dem_lev_matrix

    def dem_lev_test(self):
        self.assertEqual(self.function("aba", "aab"), 1)
        self.assertEqual(self.function("ab", "ba"), 1)
        self.assertEqual(self.function("abb", "bab"), 1)


# Алгоритмы поиска расстояния Левенштейна проходят одинковые тесты
# из класса LevTest
# Алгоритм поиска расстояния Левенштейна, матричный метод
class LevMatrixTest(LevTest):
    def setUp(self):
        self.function = main.lev_matrix
# Алгоритм поиска расстояния Левенштейна, рекурсивный метод
class LevRecursionTest(LevTest):
```

```
    def setUp(self):
        self.function = main.lev_recursion
# Алгоритм поиска расстояния Левенштейна,
# рекурсивный метод с заполнением матрицы
class LevMatRecTest(LevTest):
    def setUp(self):
        self.function = main.lev_matrix_recursion

# Точка входа, запуск тестов
if __name__ == "__main__":
    unittest.main()
```

3.4. Оценка памяти

Произведём оценку наибольшей затрачиваемой алгоритмом памяти M_{max} при поиске расстояний для строк s1 и s2. Для удобства оценки примем длину обеих строк за n.

Расстояние Левенштейна, матричный метод. Память затрачивается на матрицу и две строки.

M_{max} = (n+1)*(n+1)*sizeof(int) + (n+n)*sizeof(char) =

(n+1)*(n+1)*16 + (n+n) = 16*n^2 + 2*17n + 16 байт

Расстояние Дамерау-Левенштейна, матричный метод. Аналогично.

M_{max} = 16*n^2 + 2*17n + 16 байт

Расстояние Левенштейна, рекурсивный метод. Память используется при каждом вызове функции. Одна функция принимает в качестве аргумента 2 строки по значению, 2 размера строк. Максимальная глубина рекурсии = n+n.

M_{max} = (n+n)*(2n*sizeof(char) + 2*sizeof(int)) = 2n*(2n + 32) = 4n^2 + 64n байт.

Расстояние Левенштейна, рекурсивный метод с заполнением матрицы. Память используется для матрицы и при каждом вызове функции. Максимальная глубина рекурсии = n+n.

M_{max} = (n+1)*(n+1)*sizeof(int) + (n+n)*(2n*sizeof(char) + 2*sizeof(int)) = (n^2+2n+1)*16 + 2n*(2n + 32) = 20n^2 + 96n + 16 байт.

3.5. Среда и инструменты замера

Для замера процессорного времени исполнения функции используется библиотека time. Проведение измерений производится в функции, приведённой в листинге 3.5. Также в листинге приведена функция random_str для создания строки заданной длины из случайной последовательности символов, с использованием библиотеки random.

Листинг 3.5. Функция замера процессорного времени работы функции.

```
def random_str(length):
    a = []
    for i in range(length):
        a.append(random.choice("qwerty"))
    return "".join(a)

def test_memory(func, length):
    s1 = random_str(length)
    s2 = random_str(length)
    print("Строка 1:", s1)
    print("Строка 2:", s2)

    p = psutil.Process()
    mem1 = p.memory_info().peak_wset
    func(s1, s2)
    mem2 = p.memory_info().peak_wset

    print("Затраченая память - {:} байт".format(mem2-mem1))
```

Исследовательская часть

План экспериментов

Измерения процессорного времени проводятся при равных длинах строк s1 и s2. Содержание строк сгенерировано случайным образом. Изучается время работы при длинах: 1, 3, 10, 20, 100, 1000. Для повышения точности, каждый замер производится три раза, за результат берётся среднее арифметическое.

Результат экспериментов

По результатам измерений процессорного времени можно составить таблицу 4.1.

Таблица 4.1. Результат измерений процессорного времени (в секундах)

Длина строк / Алгоритм	1	3	10	20	100	1000
Лев., матрица	$7*10^{-6}$	$1.9*10^{-5}$	$1.3*10^{-4}$	$4.7*10^{-4}$	0.013	1.405
Лев., рекурсия	$3*10^{-6}$	$4.7*10^{-5}$	6.984	-	-	-
Лев., рекурсия с матрицей	$1*10^{-5}$	$4.1*10^{-5}$	$4.1*10^{-4}$	$2.5*10^{-3}$	0.38	-
Д-Л, матрица	$8*10^{-6}$	$2.8*10^{-5}$	$1.7*10^{-4}$	$6.1*10^{-4}$	0.016	2.031

В алгоритме нахождения расстояния Левенштейна с помощью рекурсии замеры на длине строк более 10 не проводились, так как время выполнения было слишком велико (более 10 минут). В алгоритме рекурсии с заполнением матрицы не удалось провести измерения при длине 1000, так как была превышена максимальная глубина рекурсии.

Сравнительный анализ

По результатам эксперимента можно заключить следующее.

- Наиболее быстродейственным алгоритмом поиска расстояния Левенштейна является алгоритм, использующий матрицу.
- Рекурсивный алгоритм с использованием матрицы показывает значительно более низкую скорость роста времени по сравнению с рекурсивным алгоритмом.
- Алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна с помощью матрицы показывают схожую скорость роста времени, однако первый алгоритм несколько быстрее.

Заключение

В ходе лабораторной работы достигнута поставленная цель: …. Решены все задачи работы. Были изучены и описаны понятия расстояний Левенштейна и Дамерау-Левенштейна. Также были описаны и реализованы алгоритмы поиска расстояний. Проведены замеры процессорного времени работы каждого алгоритмах при различных строках, оценена наибольшая занимаемая память. На основании оценок и экспериментов проведён сравнительный анализ.

Список литературы