МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №3
по дисциплине Построение и анализ алгоритмов
Тема: «Редакционное расстояние»

Студент гр. 3343 _____ Пивоев Н. М.

Преподаватель _____ Жангиров Т. Р.

Санкт-Петербург
2025

Цель работы

Изучить работу алгоритма Вагнера-Фишера для построения матрицы расстояния Левенштейна и нахождения редакционного предписания.

Задание 1

Над строкой ε (будем считать строкой непрерывную последовательность из латинских букв) заданы следующие операции:

1. *replace(ε, a, b)* – заменить символ *a* на символ *b*.
2. *insert(ε, a)* – вставить в строку символ *a* (на любую позицию).
3. *delete(ε, b)* – удалить из строки символ *b*.

Каждая операция может иметь некоторую цену выполнения (*положительное число*).

Даны две строки A и B, а также три числа, отвечающие за цену каждой операции. Определите минимальную стоимость операций, которые необходимы для превращения строки A в строку B.

Входные данные: первая строка – три числа: цена операции *replace*, цена операции *insert*, цена операции *delete*; вторая строка – A; третья строка – B.

Выходные данные: одно число – минимальная стоимость операций.

Sample Input:

1 1 1
entrance
reenterable

Sample Output:

5

Задание 2

Над строкой ε (будем считать строкой непрерывную последовательность из латинских букв) заданы следующие операции:

1. *replace(ε, a, b)* – заменить символ *a* на символ *b*.

2. *insert(ε, a)* – вставить в строку символ *a* (на любую позицию).

3. *delete(ε, b)* – удалить из строки символ *b*.

Каждая операция может иметь некоторую цену выполнения (*положительное число*).

Даны две строки A и B, а также три числа, отвечающие за цену каждой операции. Определите последовательность операций (редакционное предписание) с минимальной стоимостью, которые необходимы для превращения строки A в строку B.

Пример (все операции стоят одинаково)

M	**M**	**M**	**R**	**I**	**M**	**R**	**R**
C	**O**	**N**	**N**		**E**	**C**	**T**
C	O	N	**E**	**H**	E	A	D

Пример (цена замены 3, остальные операции по 1)

M	**M**	**M**	**D**	**M**	**I**	**I**	**I**	**I**	**D**	**D**
C	**O**	**N**	**N**	**E**					**C**	**T**
C	O	N		E	**H**	**E**	**A**	**D**		

Входные данные: первая строка – три числа: цена операции *replace*, цена операции *insert*, цена операции *delete*; вторая строка – A; третья строка – B.

Выходные данные: первая строка – последовательность операций (M – совпадение, ничего делать не надо; R – заменить символ на другой; I – вставить символ на текущую позицию; D – удалить символ из строки); вторая строка – исходная строка A; третья строка – исходная строка B.

Sample Input:

1 1 1

entrance

reenterable

Sample Output:

IMIMMIMMRRM

entrance

reenterable

Задание 3

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

Разработайте программу, осуществляющую поиск расстояния Левенштейна между двумя строками.

Пример:

Для строк pedestal и stien расстояние Левенштейна равно 7:

- Сначала нужно совершить четыре операции удаления символа: pedestal -> stal.
- Затем необходимо заменить два последних символа: stal -> stie.
- Потом нужно добавить символ в конец строки: stie -> stien.

Параметры входных данных:

Первая строка входных данных содержит строку из строчных латинских букв. (S, $1 \le |S| \le 2550$).

Вторая строка входных данных содержит строку из строчных латинских букв. (T, $1 \le |T| \le 2550$).

Параметры выходных данных:

Одно число L, равное расстоянию Левенштейна между строками S и T.

Sample Input:

pedestal

stien

Sample Output:

7

Вариант 1.

"Особо заменяемый символ и особо добавляемый символ": цена замены определённого символа отличается от обычной цены замены; цена добавления другого (или того же) определённого символа отличается от обычной цены добавления. Особо заменяемый символ и цена его замены, особо добавляемый символ и цена его добавления — дополнительные входные данные.

Выполнение работы

Описание алгоритма

Расстояние Левенштейна показывает, в какое количество действий с символами одно слово можно преобразовать в другое, а Алгоритм Вагнера-Фишера – это алгоритм нахождения этого расстояния путём составления матрицы расстояний.

Сначала идёт заполнение матрицы расстояний. Она строится на основе двух рассматриваемых слов. Каждое значение в ней – расстояние Левенштейна для двух подстрок, полученных путём обрезания оригинальных строк по индексам строки и столбца. Мы преобразуем первое поданное слово во второе. Рассмотрим основные случаи при заполнении:

$d[0][0] = 0$ – расстояние между двумя пустыми строками – нулевое.

$d[0][j] = j$ + *insertion_cost* (первая строка) – чтобы получить из пустой строки вторую (или её подстроку), нужно выполнить j вставок.

$d[i][0] = i$ + *deletion_cost* (первый столбец) – чтобы получить из начальной строки (или её подстроки) пустую, нужно выполнить i удалений.

$d[i][j], i > 0, j > 0$ – для остальных ячеек матрицы берётся минимум из определённых вычисленных значений + цена соответствующей операции. Среди рассматриваемых операций:

значение слева ($d[i][j-1]$) + цена вставки текущего символа,

значение сверху ($d[i-1][j]$) + цена удаление текущего символа,

значение слева-сверху ($d[i-1][j-1]$) + цена замены, если символы совпадают, то цена замены = 0.

Таким образом, значение в правом нижнем углу матрицы – расстояние Левенштейна для рассматриваемых строк.

Затем находится редакционное предписание – последовательность операций, которые преобразуют первую строку во вторую. Для этого необходимо из конечного значения матрицы (справа снизу) вернуться в начальную (слева сверху) по наиболее оптимальному пути. Берётся минимум из значения + цены

операции для левой, верхней и левой верхней ячеек, которые соответственно обозначают операции вставки, удаления и замены (в случае совпадении символов операция не требуются).

Оценка сложности

Временная сложность алгоритма – $O(n * m)$, где n – длина первой строки, а m – длина второй. Сложность квадратичная, поскольку программа составляет матрицу расстояний, размером $(n + 1) * (m + 1)$.

Пространственная сложность алгоритма – $O(n * m)$, поскольку нам необходимо хранить непосредственно матрицу размером $(n + 1) * (m + 1)$.

Код программы содержит реализацию следующих функций:

- *decide_costs(symbol_1, symbol_2, costs, special_costs = None)* – определяет, являются ли обрабатываемые символы особыми и изменяет цену операций для них.
- *get_distance_matrix(s1, s2, replacement_cost, insertion_cost, deletion_cost)* – составляет матрицу расстояний, основываясь на полученных в предыдущих шагах расстояниях и ценах операций.
- *traceback_operations(d, s1, s2, replacement_cost, insertion_cost, deletion_cost)* – обходит матрицу расстояний с правого нижнего угла к левому верхнему, получая все операции, которые ведут к достижению наименьшего расстояния Левенштейна.
- *check_solution(s1, s2, solution)* – применяет операции, полученные обратным обходом матрицы, к первой строке, так, чтобы в конце она совпала со второй строкой.
- *def get_costs(is_special)* – обрабатывает ввод цен.

Тестирование

Программа была протестирована на различных входных данных. Составлена соответствующая таблица:

Таблица 1.

Входные данные	Выходные данные	Комментарий
1 1 1 abc abcdefg	MMMIIII 4	Добавление символов
1 1 1 abcdefg abc	MMMDDDD 4	Удаление символов
1 1 1 abcaaaa abcdefg	MMMRRRR 4	Замена символов
1 1 1 кот собака	RMIIIR 5	Полноценная работа
1 1 1 connect conehead	MMMIRMRR 4	Полноценная работа
1 1 3 собака кот	RMDDDR 11	Разные цены операций
3 1 1 connect conehead	MMDMMDDIIII 7	Разные цены операций. Удаление со вставкой выгоднее замены

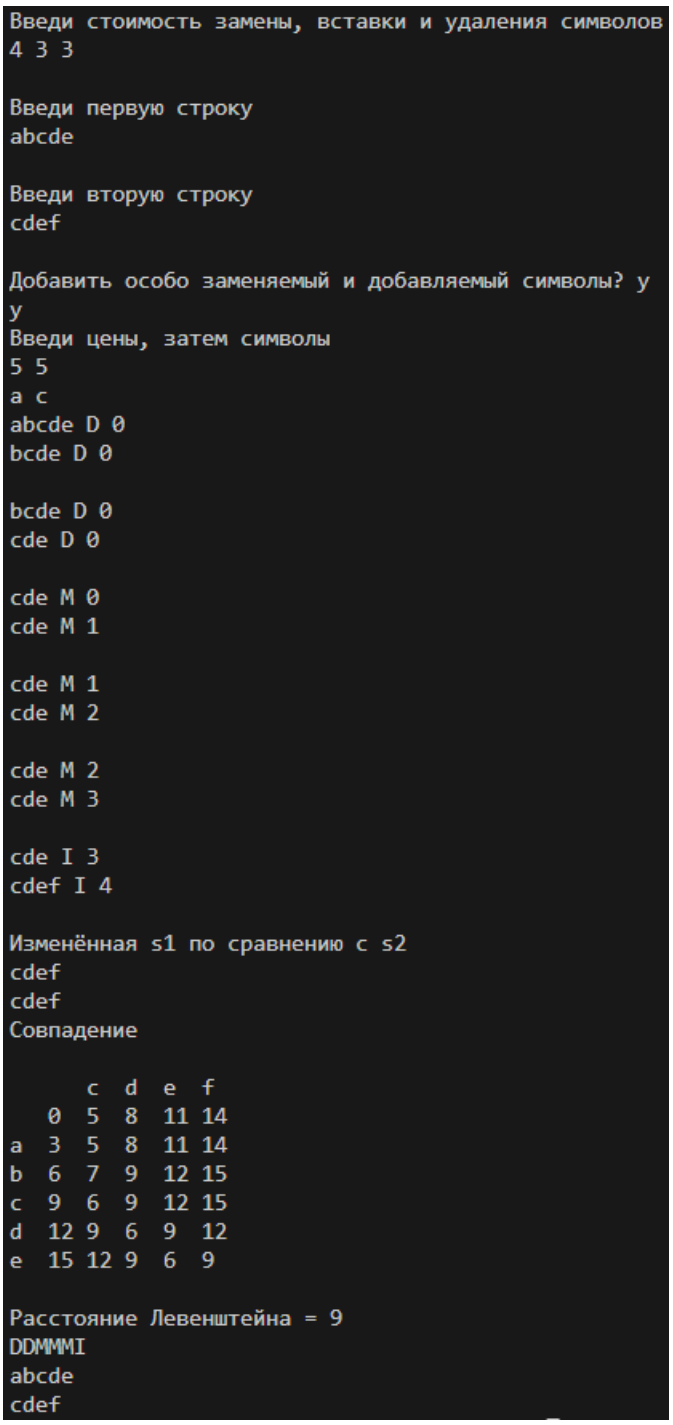

Рисунок 1 – Результат работы программы.

Исследование

Исследуем эффективность алгоритма Вагнера-Фишера для построения матрицы расстояний на различных объёмах входных данных.

Таблица 3. Исследование эффективности по времени.

Размер первого слова	Размер второго слова	Произведение размеров	Затраченное время (с)
10	10	100	0.00005
10	1.000	10.000	0.00408
1.000	10	10.000	0.00492
100	10.000	1.000.000	0.42381
10.000	100	1.000.000	0.43473
1.000	1.000	1.000.000	0.40937
1.000.000	10	10.000.000	5.64362

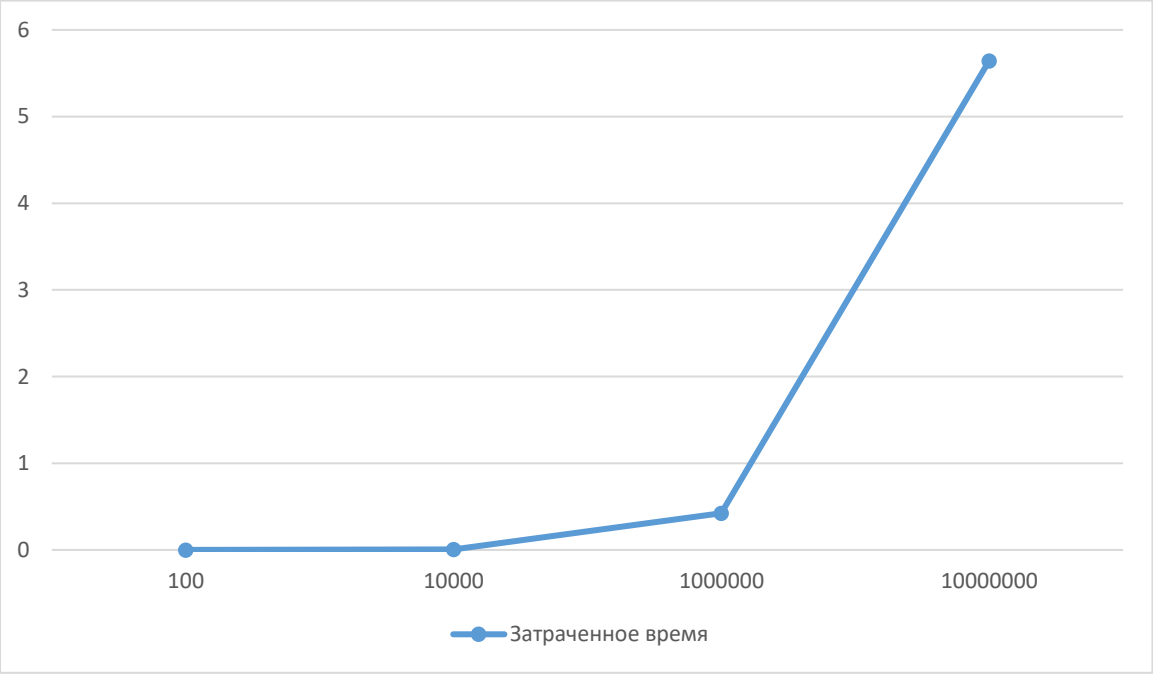

Рисунок 3 – График зависимости затраченного времени на получение матрицы расстояний от произведения размеров входных строк.

Можно сделать следующие выводы по исследованию:

1. Полученные результаты подтверждают временную оценку $O(n * m)$.
2. Результат по времени не зависит от того, какая из строк длиннее.

Выводы

Во время выполнения лабораторной работы, была изучена работа алгоритма Вагнера-Фишера. Решены задачи поиска матрицы расстояния Левенштейна и нахождения редакционного предписания.

ПРИЛОЖЕНИЕ
ИСХОДНЫЙ КОД ПРОГРАММЫ

Имя файла: main.py

```
from levenshtein import *


def get_costs(is_special):
    try:
        costs = list(map(int, input().split()))
        if is_special:
            if len(costs) != 2:
                raise ValueError
        else:
            if len(costs) != 3:
                raise ValueError

        return costs
    except:
        print("Стоимости некорректны")
        exit(1)


def main():
    print("Введи стоимость замены, вставки и удаления символов")
    costs = Costs(*get_costs(0))

    print("\nВведи первую строку")
    s1 = input()

    print("\nВведи вторую строку")
    s2 = input()
    print()

    print("Добавить особо заменяемый и добавляемый символы? y")
    if input() == 'y':
        print("Введи цены, затем символы")
        special_costs = Special_Costs(*get_costs(1), *input().split())

        d = get_distance_matrix(s1, s2, costs, special_costs)
        solution = traceback_operations(d, s1, s2, costs, special_costs)
    else:
        d = get_distance_matrix(s1, s2, costs)
        solution = traceback_operations(d, s1, s2, costs)

    check_solution(s1, s2, solution)

    print('    ', *[c for c in s2], sep='  ')
    for i, column in enumerate(d):
        if i == 0:
            print(' ', end='  ')
        else:
            print(s1[i - 1], end='  ')

        for j in column:
            if j >= 10:
                print(j, end=' ')
                continue
            print(j, end='  ')
        print()
```

```
        print()

        print("Расстояние Левенштейна =", d[len(s1)][len(s2)])
        print(solution, s1, s2, sep='\n')


    if __name__ == "__main__":
    main()
```

Имя файла: levenshtein.py

```
    class Costs:
        def __init__(self, replacement_cost, insertion_cost, deletion_cost):
            self.replacement_cost = replacement_cost
            self.insertion_cost = insertion_cost
            self.deletion_cost = deletion_cost


    class Special_Costs:
        def __init__(self, replacement_cost, insertion_cost, replacement_symbol,
insertion_symbol):
            self.replacement_cost = replacement_cost
            self.insertion_cost = insertion_cost
            self.replacement_symbol = replacement_symbol
            self.insertion_symbol = insertion_symbol


    def decide_costs(symbol_1, symbol_2, costs, special_costs = None):
        if special_costs and symbol_1 == special_costs.replacement_symbol:
            replacement_cost = special_costs.replacement_cost
        else:
            replacement_cost = costs.replacement_cost

        if special_costs and symbol_2 == special_costs.insertion_symbol:
            insertion_cost = special_costs.insertion_cost
        else:
            insertion_cost = costs.insertion_cost
        return replacement_cost, insertion_cost


    def get_distance_matrix(s1, s2, costs, special_costs = None):
        m, n = len(s1), len(s2)

        d = [[-1] * (n+1) for _ in range(m+1)]
        d[0][0] = 0

        for j in range(1, n + 1):
            if special_costs and s2[j - 1] == special_costs.insertion_symbol:
                d[0][j] = d[0][j - 1] + special_costs.insertion_cost
                continue
            d[0][j] = d[0][j - 1] + costs.insertion_cost

        for i in range(1, m + 1):
            d[i][0] = d[i - 1][0] + costs.deletion_cost
            for j in range(1, n + 1):
                if s1[i-1] == s2[j-1]:
                    d[i][j] = d[i - 1][j - 1]
                    continue

                replacement_cost, insertion_cost = decide_costs(s1[i - 1], s2[j
- 1], costs, special_costs)
```

```
                d[i][j] = min(
                    d[i - 1][j - 1] + replacement_cost,
                    d[i][j - 1] + insertion_cost,
                    d[i - 1][j] + costs.deletion_cost
                )
        return d


    def traceback_operations(d, s1, s2, costs, special_costs = None):
        m, n = len(s1), len(s2)
        solution = ''
        i, j = m, n
        while i > 0 or j > 0:
            if i > 0 and j > 0 and s1[i - 1] == s2[j - 1]:
                solution += "M"
                i -= 1
                j -= 1
                continue

            replacement_cost, insertion_cost = decide_costs(s1[i - 1], s2[j -
1], costs, special_costs)

            if i > 0 and j > 0 and d[i - 1][j - 1] + replacement_cost == d[i][j]:
                solution += 'R'
                i -= 1
                j -= 1
            elif j > 0 and d[i][j - 1] + insertion_cost == d[i][j]:
                solution += 'I'
                j -= 1
            elif i > 0 and d[i - 1][j] + costs.deletion_cost == d[i][j]:
                solution += 'D'
                i -= 1
        return solution[::-1]


    def check_solution(s1, s2, solution):
        s = list(s1)
        ptr = 0
        for option in solution:
            print(''.join(s), option, ptr)
            if option == 'R':
                s[ptr] = s2[ptr]
            elif option == 'I':
                s.insert(ptr, s2[ptr])
            elif option == 'D':
                del s[ptr]
                ptr -= 1
            ptr += 1
            print(''.join(s), option, ptr, "\n")

        print("Изменённая s1 по сравнению с s2")
        print(''.join(s))
        print(s2)
        if ''.join(s) == s2:
            print("Совпадение\n")
```