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 _____ Лихацкий В. Р.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2025

Цель работы.

Написании программы вычисления редакционного расстояния и предписания алгоритмом Вагнера-Фишера.

Задание.

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*S*, 1≤|S|≤25501≤|*S*|≤2550).

Вторая строка входных данных содержит строку из строчных латинских букв. (T*T*, 1≤|T|≤25501≤|*T*|≤2550).

Параметры выходных данных:

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

Sample Input:

pedestal

stien

Sample Output:

7

Вариант 6б. Добавляется 4-я операция со своей стоимостью: одновременная замена двух последовательных символов, при этом ни один из символом

заменённой пары не должен быть в новой паре (ни на своём месте, ни на месте "партнёра").

Описаниеалгоритма.

Алгоритм Вагнера-Фишера — это алгоритм динамического программирования, предназначенный для вычисления редакционного расстояния (расстояния Левенштейна) между двумя строками. Это минимальное количество операций вставки, удаления или замены символов, необходимых для преобразования одной строки в другую.

Основные шаги алгоритма:

1)Инициализация матрицы:
Создаётся матрица размером (n+1) × (m+1), где n и m — длины строк.
Первая строка заполняется числами от 0 до n (стоимость удаления символов).
Первый столбец заполняется числами от 0 до m (стоимость вставки символов).
2) Заполнение матрицы:
Для каждой ячейки [i][j] (где i — символ первой строки, j — второй) вычисляется минимальная стоимость операций:
Удаление: cost = matrix[i-1][j] + 1
Вставка: cost = matrix[i][j-1] + 1
Замена:
Если символы совпадают (str1[i-1] == str2[j-1]), то замена бесплатна: cost = matrix[i-1][j-1].
Иначе: cost = matrix[i-1][j-1] + 1.
Выбирается минимальное из трёх значений.
3) Результат:
Значение в правом нижнем углу матрицы (matrix[n][m]) — искомое редакционное расстояние.

4) Сложность:

По времени: **O(n*m)**, т.к. алгоритм заполняет матрицу размером (n+1) * (m+1), где: n — длина первой строки,m — длина второй строки.Пространственная: **O(n·m),** т.к для хранения матрицы потребуется память пропорциональная n*m.

Алгоритм поиска редакционного предписания

1) Построение матрицы расстояний

Создаётся матрица D размером (n+1) × (m+1), где n и m — длины
строк str1 и str2.

D[i][j] — расстояние между подстроками str1[0..i-1] и str2[0..j-1].

Заполнение матрицы аналогично алгоритму Вагнера-Фишера.

2) Восстановление операций

Начиная с ячейки D[n][m], двигаемся к D[0][0], выбирая путь с минимальной стоимостью.

Для каждой ячейки D[i][j] определяем, какая операция была применена:

Шаг вверх (D[i-1][j] → D[i][j]): удаление символа str1[i-1].

Шаг влево (D[i][j-1] → D[i][j]): вставка символа str2[j-1].

Шаг по диагонали (D[i-1][j-1] → D[i][j]):

Если str1[i-1] == str2[j-1]: совпадение (операция не требуется).

Иначе: замена str1[i-1] на str2[j-1].

Описание функций и структур данных.

- *Class Levenshtein*
  - *initMatrix* – заполняет верхнюю строку операциями вставки, а левый столбец операциями удалени*я*
  - *fillMatrix* – последовательно заполняет оставшиеся ячейки матрицы на каждом шагу выбирая наиболее дешевую
  - *distance* – выводит редакционное расстояние для заданных src, dest
  - *steps* – выводит редакционное предписание для заданных src, dest.
  - *Operations* - содержит основные операции (вставка, замена, удаление, совпадение) и дополнительные, объявленные пользователем
- *Type Operation*
  - *Name – название операции*
  - *Cost – стоимость операции*
  - *Call – рассчитывает итоговую стоимость операции для данной ячейки матрицы*

  Любой тип удовлетворяющий типу Operation можно использовать как новую операцию, для моего варианта это doubleReplace.

Визуализация

Для данной лабораторной работы была написана визуализация с графическим интерфейсом, в которой можно изменять начальное и итоговое слова и стоимости операций

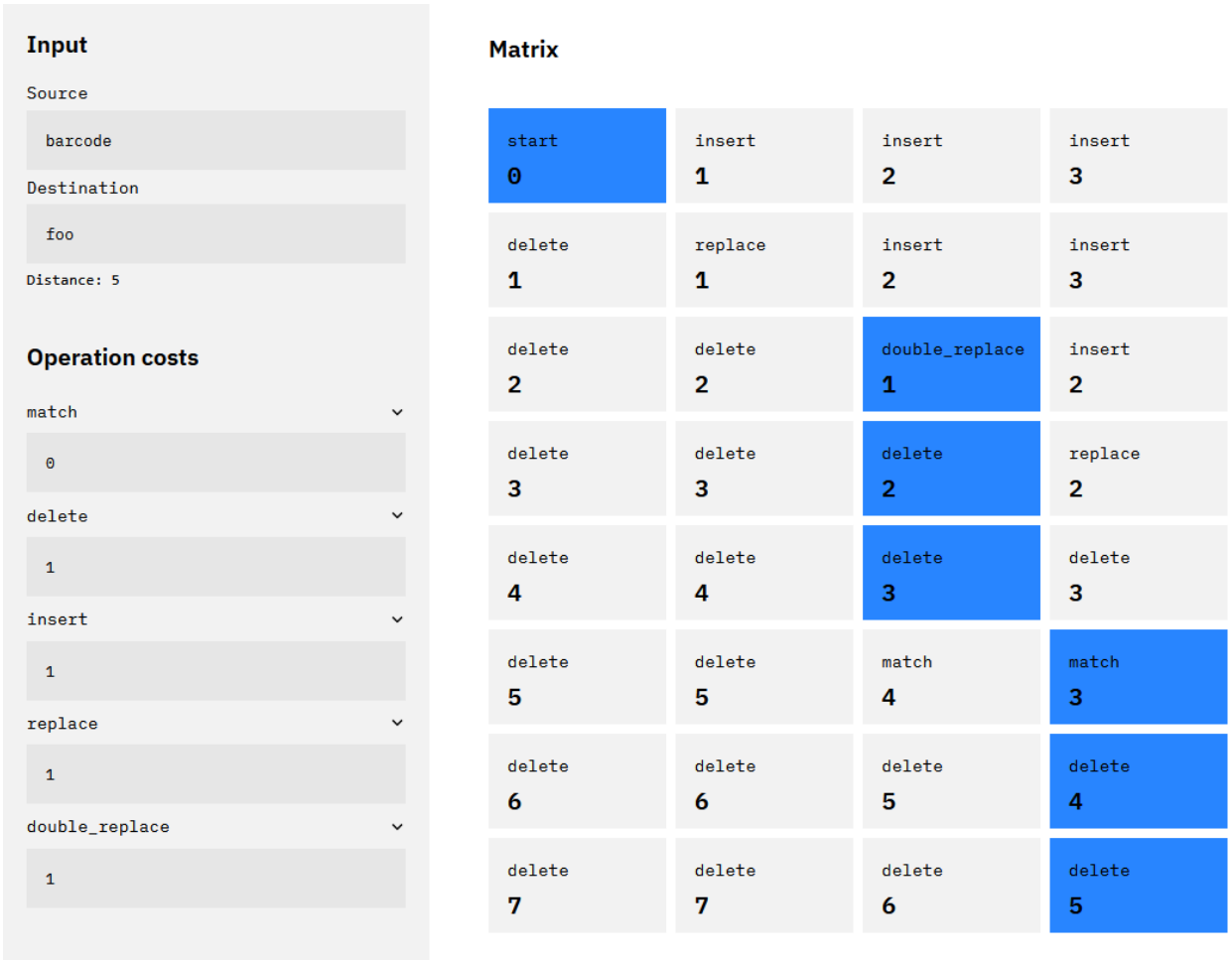

Рисунок 1 – Пример визуализации алгоритма

Тестирование.

Таблица 1 – Результаты тестирования для первого задания

№	Входные данные	Выходные данные	Комментарий
1	ab abfagfab	6	Верно
2	hello world	4	Верно
3	ВАСИЛЬЕВ КВАСЮТИН	6	Верно
4	pedestal stien	7	Верно
5	connect conehead	4	Верно

Выводы.

Был реализован алгоритм Вагнера-Фишера для вычисления редакционного расстояния и предписания между двумя строками, определяя минимальное количество операций (вставки, удаления, замены) для преобразования одной строки в другую. Алгоритм эффективно решает задачи сравнения строк, исправления опечаток и других приложений, связанных с обработкой текста.

ПРИЛОЖЕНИЕА

ИСХОДНЫЙКОДПРОГРАММЫ

Имяфайла: levenshtein.ts

```
export type Cost = number;

export type Matrix<T> = T[][];

export type Operation = {
    name: string,
    cost: Cost,
    call: (matrix: Matrix<Step>, i: number, j: number, a: string, b:
string) => {
        cost: Cost,
        i: number,
        j: number
    },
};

export type Step = {
    operation: Operation,
    cost: number,
    i: number,
    j: number,
    currentState: string,
}

export type LevenshteinParams = {
    costDelete?: Cost,
    costInsert?: Cost,
    costReplace?: Cost,
    additionalOperations?: Operation[],
}

export class Levenshtein {
    public delete: Operation = {
        name: "delete",
        cost: 1,
        call: function(matrix, i, j, a, b) {
            return {
                cost: matrix[i - 1][j].cost + this.cost,
                i: i - 1,
                j: j
            }
        }
    };
    public insert: Operation = {
        name: "insert",
        cost: 1,
        call: function(matrix, i, j, a, b) {
            return {
                cost: matrix[i][j - 1].cost + this.cost,
                i: i,
                j: j - 1
            }
        }
    };
```

```
public replace: Operation = {
    name: "replace",
    cost: 1,
    call: function(matrix, i, j, a, b) {
        return {
            cost: matrix[i - 1][j - 1].cost + this.cost,
            i: i - 1,
            j: j - 1
        }
    }
};
public match: Operation = {
    name: "match",
    cost: 0,
    call: function(matrix, i, j, a, b) {
        if(a[i - 1] == b[j - 1]) {
            return {
                cost: matrix[i - 1][j - 1].cost + this.cost,
                i: i - 1,
                j: j - 1
            }
        }

        return {
            cost: Infinity,
            i: i - 1,
            j: j - 1
        };
    }
}
public readonly additionalOperations: Operation[] = [];

public operations: Operation[] = [
    this.match,
    this.delete,
    this.insert,
    this.replace,
    ...this.additionalOperations
]

constructor({
    costDelete,
    costInsert,
    costReplace,
    additionalOperations
}: LevenshteinParams) {
    this.delete.cost = costDelete ?? 1;
    this.insert.cost = costInsert ?? 1;
    this.replace.cost = costReplace ?? 1;
    this.additionalOperations = additionalOperations ?? [];
    this.operations.push(...this.additionalOperations)
}

public matrix(a: string, b: string): Matrix<Step> {
    return this.fillMatrix(a, b);
}

public steps(a: string, b: string): Step[] {
```

```
        const matrix = this.fillMatrix(a, b)
        const steps = []

        let i = a.length;
        let j = b.length;

        while(i != 0 || j != 0) {
            const step = matrix[i][j]

            steps.push(step)

            i = step.i
            j = step.j
        }

        return steps.reverse()
    }

    public distance(a: string, b: string): Cost {
        return this.fillMatrix(a, b)[a.length][b.length].cost
    }

    private fillMatrix(a: string, b: string): Matrix<Step> {
        const matrix: Matrix<Step> = this.initMatrix(a.length, b.length);

        for(let i = 1; i < a.length + 1; i++) {
            for(let j = 1; j < b.length + 1; j++) {
                const steps: Step[] = this.operations.map(op => ({
                    ...op.call(matrix, i, j, a, b),
                    operation: op,
                    currentState: "",
                }))

                let bestStep = steps[0]
                for(let step of steps) {
                    if(step.cost < bestStep.cost) {
                        bestStep = step;
                    }
                }

                matrix[i][j] = bestStep
            }
        }

        return matrix
    }

    private initMatrix(m: number, n: number): Matrix<Step> {
        const matrix: Matrix<Step> = Array.from(
            new Array(m + 1),
            () => Array.from(new Array(n + 1))
        )

        this.delete = this.operations.find(e => e.name == "delete") ??
this.delete;
        this.insert = this.operations.find(e => e.name == "insert") ??
this.insert;
```

```
        for(let i = 0; i < m + 1; i++) {
            matrix[i][0] = {
                operation: this.delete,
                cost: i * this.delete.cost,
                i: i - 1,
                j: 0,
                currentState: ""
            };
        }

        for(let j = 0; j < n + 1; j++) {
            matrix[0][j] = {
                operation: this.insert,
                currentState: "",
                cost: j * this.insert.cost,
                i: 0,
                j: j - 1
            }
        }

        return matrix;
    }
}
```