Содержание

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 играет важную роль в определении схожести между двумя строками.

Расстояние Левенштейна применяется в теории информации и компьютерной лингвистике для решения следующих задач:

— исправление ошибок в слове(в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи);

— сравнение текстовых файлов утилитой diff;

— для сравнения геномов, хромосом и белков в биоинформатике.

Целью данной лабораторной работы является описание и исследование алгоритмов для вычисления расстояния Левенштейна и Дамерау-Левенштейна, оценка их реализаций.

Для достижения данной цели необходимо выполнить следующие задачи:

1) Описать алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейн

2) Разработать программное обеспечение, включающее в себя следующие алгоритмы:

   — нерекурсивный алгоритм для вычисления расстояния Левенштейна;

   — нерекурсивный алгоритм для вычисления расстояния Дамерау-Левенштейна;

   — рекурсивный алгоритм для вычисления расстояния Дамерау-Левенштейна;

   — рекурсивный алгоритм для вычисления расстояния Дамерау-Левенштейна с кешированием.

3) Выбрать инструменты для измерения процессорного времени выполнения реализаций алгоритмов.

4) Провести анализ затрат по времени и памяти для каждой реализации.

1 Аналитическая часть

В этом разделе будут представлены описания алгоритмов нахождения редакторских расстояний Левенштейна и Дамерау-Левенштейна.

1.1 Расстояние Левенштейна

Расстояние Левенштейна [1] (редакционное расстояние, дистанция редактирования) — это минимальное количество редакторских операций вставки (I, от англ. insert), замены (R, от англ. replace) и удаления (D, от англ. delete), необходимых для преобразования одной строки в другую. Стоимость операции зависит от ее вида:

1) $w(a, b)$ — цена замены символа a на b;

2) $w(\lambda, b)$ — цена вставки символа b;

3) $w(a, \lambda)$ — цена удаления символа a.

Пусть стоимость каждой вышеизложенной операции равной 1:

— $w(a, b) = 1$, $a \neq b$, в противном случае замена не происходит;

— $w(\lambda, b) = 1$;

— $w(a, \lambda) = 1$.

Введем понятие совпадения символов — M (от англ. match). Его стоимость будет равна 0, то есть $w(a, a) = 0$.

Введем в рассмотрение функцию $D(i, j)$, значением которой является редакционное расстояние между подстроками $S_1[1...i]$ и $S_2[1...j]$.

Расстояние Левенштейна между двумя строками S_1 и S_2 длиной M

и N соответственно рассчитывается по рекуррентной формуле:

$$D(i,j) = \begin{cases} 0, & \text{i} = 0, \text{j} = 0 \\ i, & \text{j} = 0, \text{i} > 0 \\ j, & \text{i} = 0, \text{j} > 0 \\ min \begin{cases} D(i,j-1)+1, \\ D(i-1,j)+1, \\ D(i-1,j-1)+m(S_1[i],S_2[j]), \end{cases} & \text{i} > 0, \text{j} > 0 \end{cases} \tag{1.1}$$

где сравнение символов строк S_1 и S_2 определенно как:

$$m(a,b) = \begin{cases} 0 & \text{если a} = \text{b}, \\ 1 & \text{иначе.} \end{cases} \tag{1.2}$$

1.1.1 Нерекурсивный алгоритм нахождения расстояния Левенштейна

Рекурсивный способ реализации алгоритма Левенштейна становится неэффективным с увеличением размеров строк M и N из-за большого числа промежуточных вычислений. Для повышения эффективности можно воспользоваться итеративной реализацией, где матрица промежуточных значений $D(i,j)$ заполняется пошагово.

Для хранения этих промежуточных значений можно использовать матрицу размером:

$$(N+1) \times (M+1) \tag{1.3}$$

Значения в ячейке $[i,j]$ соответствует значению $D(S1[1...i], S2[1...j])$. Первый элемент матрицы инициализируется нулем, а затем матрица заполняется в соответствии с формулой (1.1).

Тем не менее, матричный метод требует больше памяти по сравнению с рекурсивным при больших значениях M и N, поскольку он хранит все промежуточные значения $D(i,j)$. Для оптимизации использования памяти в рекурсивном алгоритме вычисления расстояния Левенштейна можно

применить кэширование. Это означает, что мы будем хранить пару строк, содержащих значения $D(i,j)$, вычисленные на предыдущей итерации алгоритма, а также значения $D(i,j)$, вычисленные на текущей итерации.

1.2 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна — это мера разницы двух строк символов, определяемая как минимальное количество операций вставки, удаления, замены и транспозиции (перестановки двух соседних символов), необходимых для перевода одной строки в другую. То есть оно является модификацией расстояния Левенштейна: к трем базовым операциям добавляется операция транспозиции T (от англ. transposition).

Расстояние Дамерау-Левенштейна может быть вычислено по рекуррентной формуле:

$$D(i,j) = \begin{cases} 0, & \text{i} = 0, \text{j} = 0, \\ i, & \text{j} = 0, \text{i} > 0, \\ j, & \text{i} = 0, \text{j} > 0, \\ min \begin{cases} D(i,j-1)+1, \\ D(i-1,j)+1, \\ D(i-1,j-1)+m(S_1[i],S_2[j]), \\ D(i-2,j-2)+1, \end{cases} & \text{если i} > 1, \text{j} > 1, \\ & S_1[i] = S_2[j-1], \\ & S_1[i-1] = S_2[j], \\ min \begin{cases} D(i,j-1)+1, \\ D(i-1,j)+1, \\ D(i-1,j-1)+m(S_1[i],S_2[j]), \end{cases} & \text{иначе.} \end{cases} \tag{1.4}$$

1.2.1 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна

Рекурсивный алгоритм реализует формулу (1.4), функция D составлена таким образом, что верно следующее.

1) Для передачи из пустой строки в пустую требуется ноль операций.

2) Для перевода из пустой строки в строку a требуется $|a|$ операций.

3) Для перевода из строки a в пустую строку требуется $|a|$ операций.

4) Для перевода из строки a в строку b требуется выполнить последовательно некоторое количество операций удаления, вставки, замены, транспозиции в некоторой последовательности. Последовательность поведения любых двух операций можно поменять, порядок поведения операций не имеет никакого значения. Пусть *a', b'* – строки a и b без последнего символа соответственно, а *a", b"* – строки a и b без двух последних символов. Тогда цена преобразования из строки a в b может быть выражена как:

   — сумма цены преобразования строки a' в b и цены проведения операции удаления, которая необходима для преобразования a' в a;

   — сумма цены преобразования строки a в b' и цены проведения операции вставки, которая необходима для преобразования b' в b;

   — сумма цены преобразования из a' в b' и операции замены, предполагая, что a и b оканчиваются на разные символы;

   — сумма цены преобразования из a'' в b'' и операции перестановки, предполагая, что длины a'' и b'' больше 1 и последние два символа a'', поменянные местами, совпадут с двумя последними символами b'';

   — цена преобразования из a' в b', предполагая, что a и b оканчиваются на один и тот же символ.

Минимальной стоимостью преобразования будет минимальное значение приведенных вариантов.

1.2.2 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна с кешированием

Рекурсивная версия алгоритма Дамерау-Левенштейна становится неэффективной при больших значениях N и M из-за повторного вычисления расстояний между подстроками. Для оптимизации этого алгоритма мы можем использовать матрицу для хранения промежуточных значений. В этой версии алгоритма матрица $A_{|a|,|b|}$ используется для хранения значений $D(i, j)$, что позволяет избежать повторных вычислений и делает эту матрицу своего рода кешем для рекурсивной реализации.

1.2.3 Нерекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна

Рекурсивная реализация алгоритма Левенштейна с применением кеширования оказывается малоэффективной по времени в случае больших значений M и N. Для повышения эффективности можно перейти к итеративной реализации, где матрица для хранения промежуточных значений $D(i, j)$ заполняется пошагово.

Для хранения этих значений мы можем использовать матрицу с размерами:

$$(N + 1) \times (M + 1), \tag{1.5}$$

Значение в ячейке $[i, j]$ соответствует значению $D(S1[1...i], S2[1...j])$. Первый элемент матрицы инициализируется нулем, а затем мы заполняем всю таблицу в соответствии с формулой (1.4).

Вывод

В данном разделе были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна, который является модификаций первого, учитывающего возможность перестановки соседних символов. Формулы Левенштейна и Дамерау — Левенштейна для расчета расстояния между строками задаются рекурсивно, а следовательно, алгоритмы могут быть реализованы рекурсивно или итерационно.

2 Конструкторская часть

В данном разделе будут приведены схемы алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна, описание используемых типов данных и требования к программному обеспечению.

2.1 Требования к программному обеспечению

К программе предъявлен ряд требований:

— принимает на вход две строки;

— выдает редакторское расстояние Левенштейна и Дамерау–Левенштейна;

— имеет интерфейса для выбора действий;

— может обрабатывать строки, включающие буквы как на латинице, так и на кириллице;

— имеет функциональность замера процессорного времени работы реализаций алгоритмов.

2.2 Разработка алгоритмов

На вход алгоритмов подаются строки S_1 и S_2.

На рисунке 2.1 представлена схема алгоритма поиска расстояния Левенштейна. На рисунках 2.2 – 2.5 представлены схемы алгоритмов поиска Дамерау-Левенштейна.

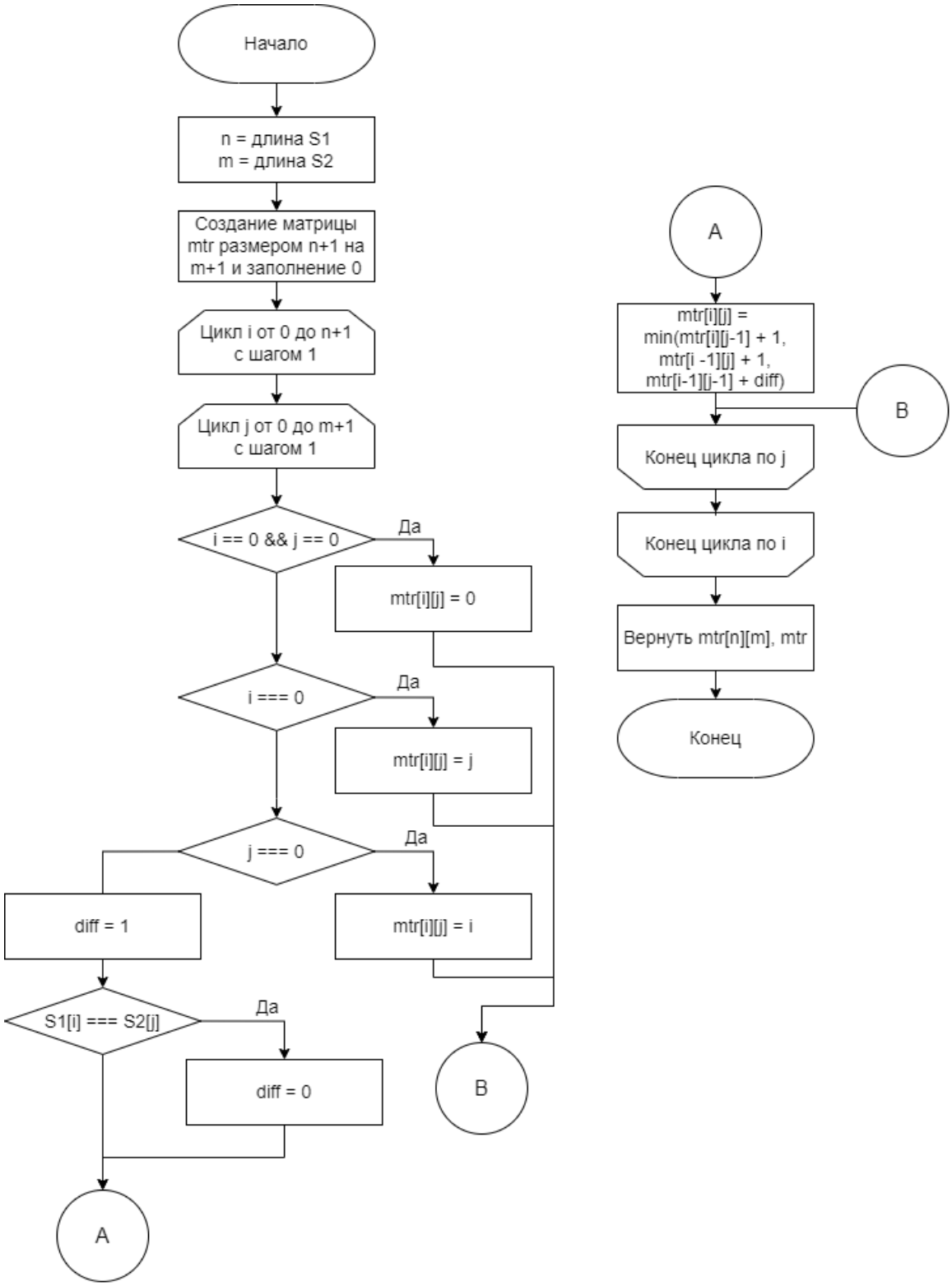

Рисунок 2.1 – Схема нерекурсивного алгоритма нахождения расстояния Левенштейна

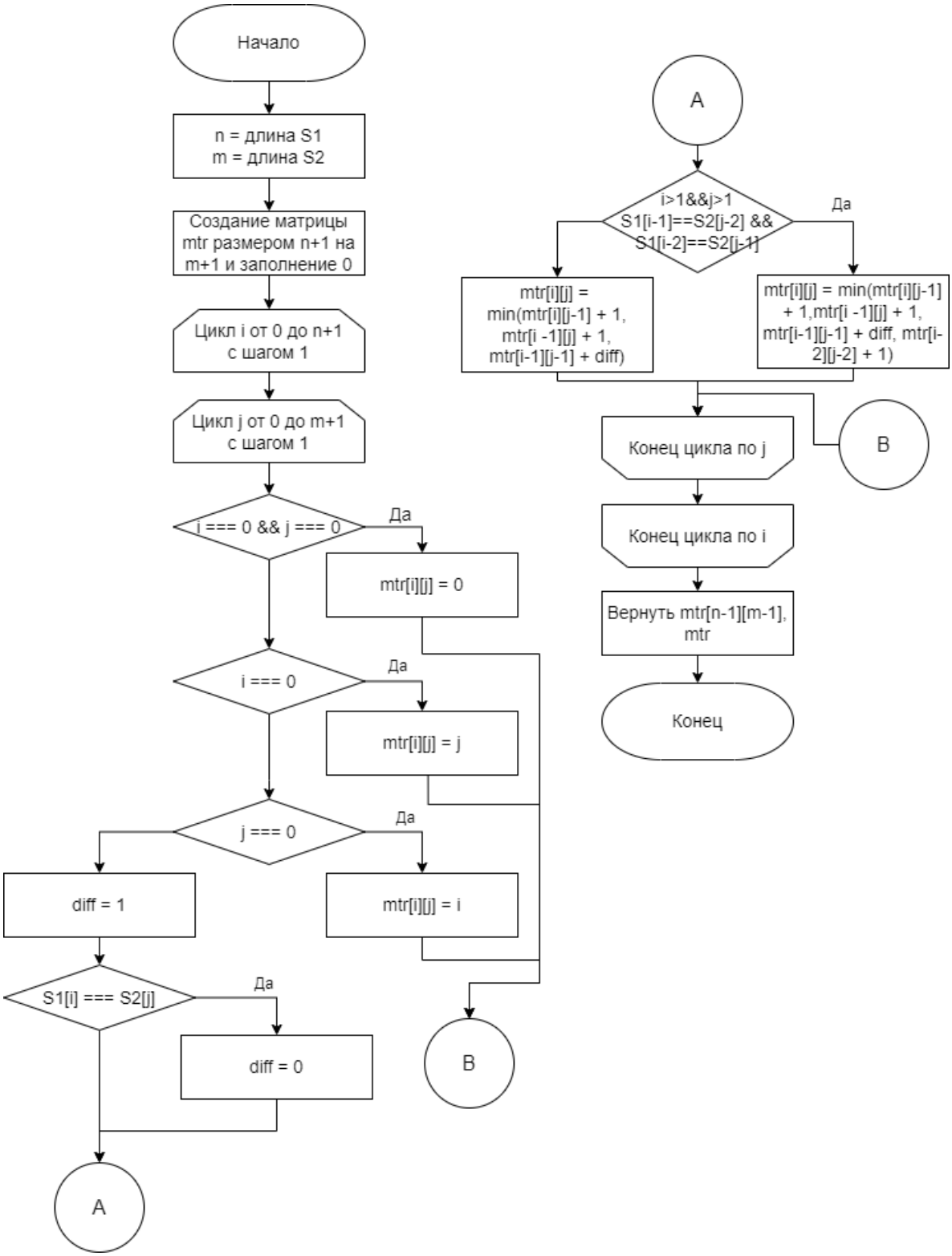

Рисунок 2.2 – Схема нерекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

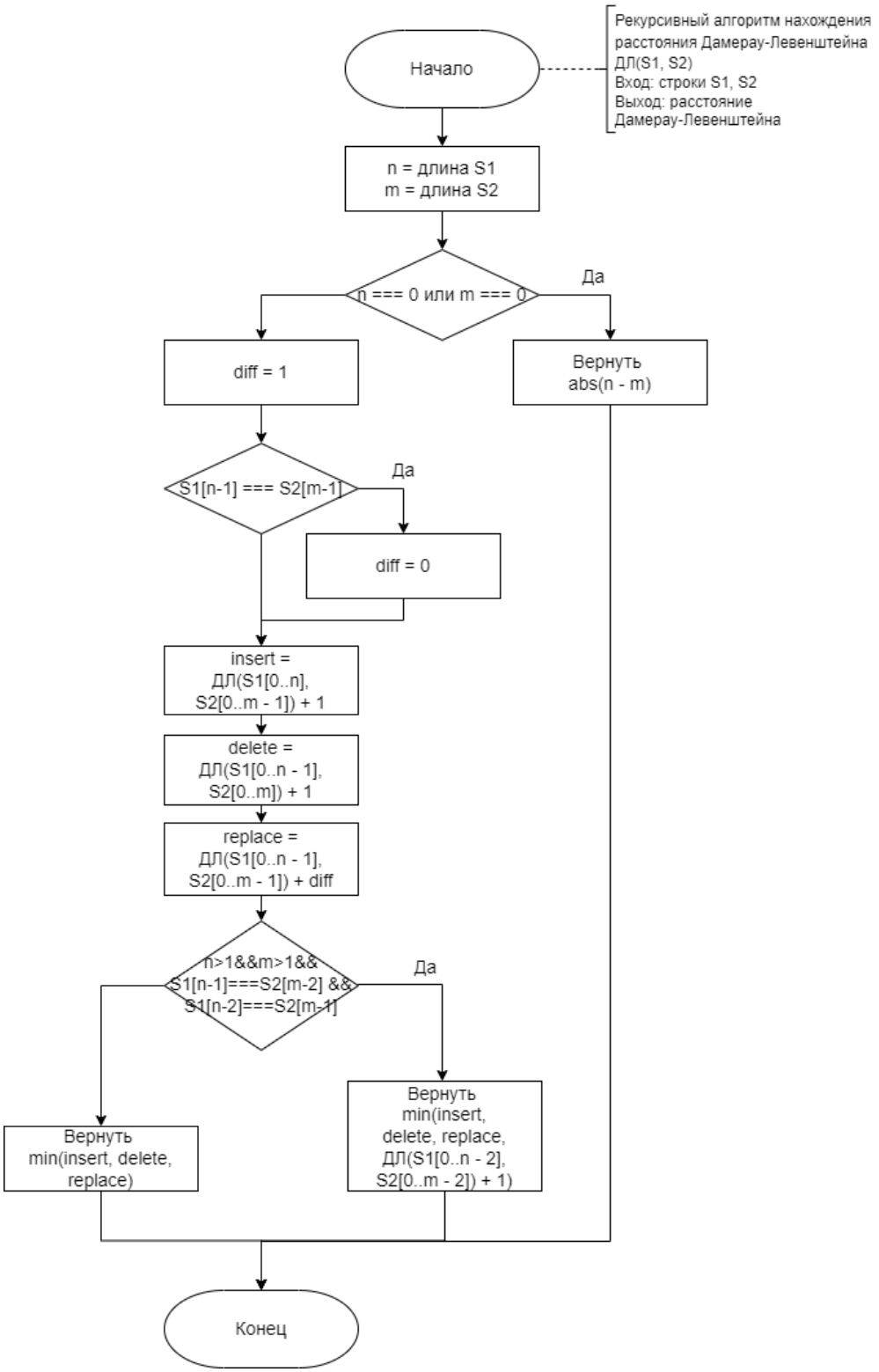

Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

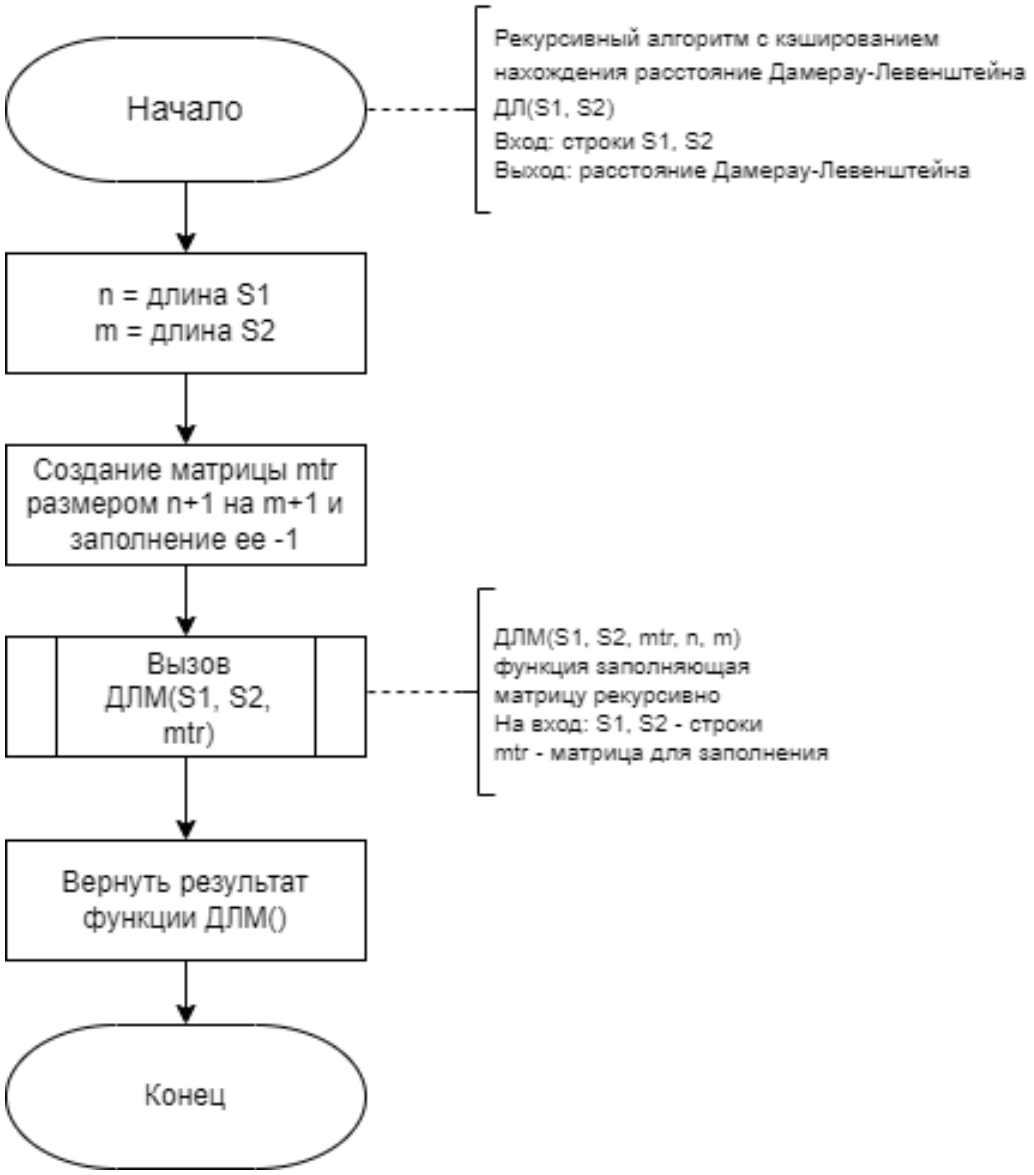

Рисунок 2.4 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с кешированием

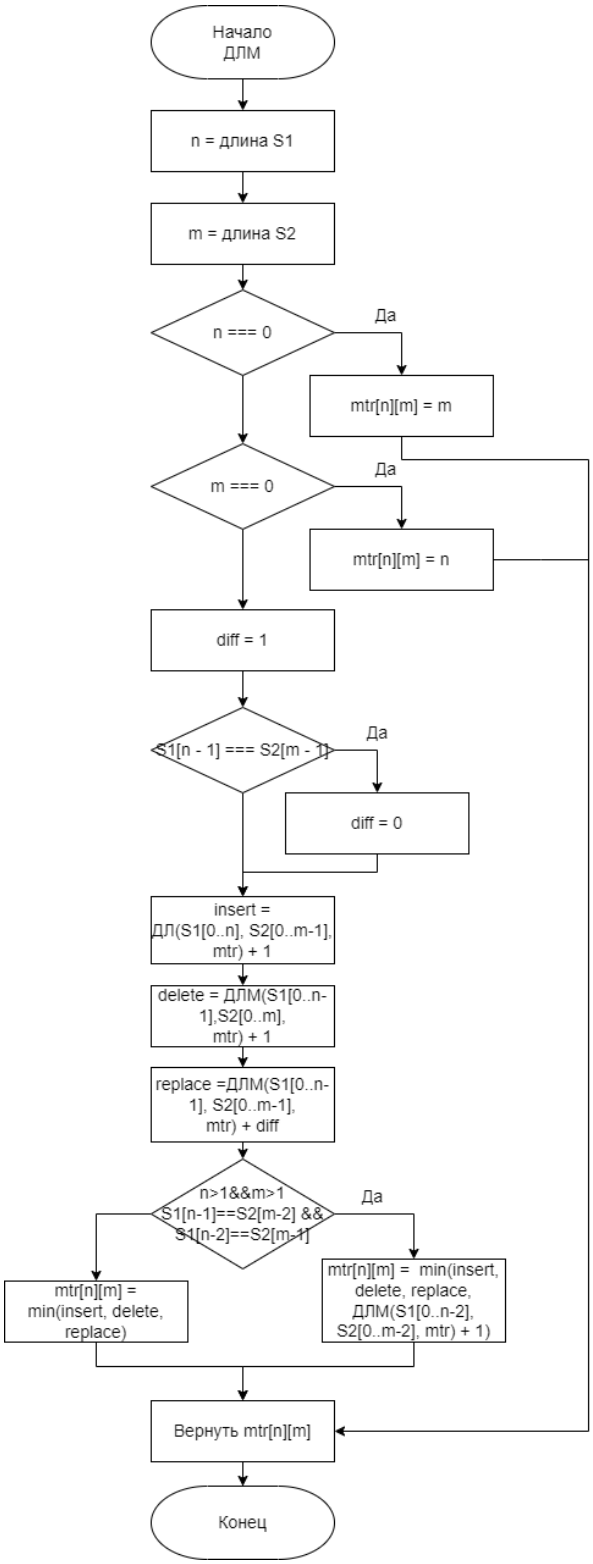

Рисунок 2.5 – Схема алгоритма рекурсивного заполнения матрицы путем поиска расстояния Дамерау-Левенштейна

2.3 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие структуры данных:

— строка — строковый литерал типа *string*;

— матрица — двумерный массив значений типа *number*.

Вывод

В данном разделе на основе теоретических данных были построены схемы требуемых алгоритмов, выбраны используемые типы данных, описаны требования к программному обеспечению.

3 Технологическая часть

В данном разделе будут описаны средства реализации, листинг кода и функциональные тесты.

3.1 Средства реализации

Для реализации ПО был выбран язык *JavaScript* [2]. Данный выбор обусловлен моим желанием расширить свои знания в области применения данного языка программирования. Также в данном языке есть все требующиеся инструменты для выполнения данной лабораторной работы.

Время работы было замерено с помощью функции *process.cpuUsage()* из программной платформы *Node.js* [3]

3.2 Сведения о модулях программы

Данная программа разбита на следующие модули.

— `main.js` — файл, содержащий точку входа в программу, из которой происходит запуск работы алгоритмов и вывод графиков.

— `algorithms.js` — файл содержит функции поиска расстояния Левенштейна и Дамерау-Левенштейна.

— `times.cjs` — файл содержит функции, замеряющие процессорное время алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна.

— `index.html` — файл веб-страницы, который используется для отображения результата работы алгоритмов.

— `style.css` — файл, который используется для определения стилей интерфейса веб-страницы.

— `measures.json` — файл, в который `times.cjs` записывает результаты замеров работы алгоритмов, и из которого впоследствии считывает данные файл `main.js` для построения графиков.

3.3 Реализация алгоритмов

В листинге 3.1 приведена реализация алгоритма создания матрицы. В листингах 3.2 – 3.5 приведены реализации алгоритмов поиска расстояний Левенштейна (нерекурсивный алгоритм) и Дамерау-Левенштейна (нерекурсивный, рекурсивный и рекурсивный с кешированием).

Листинг 3.1 – Функция создания матрицы для алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна

```
1 function createMatrix(matrix, n, m, fill) {
2     for (let i = 0; i < n; i++) {
3         matrix[i] = [];
4         for (let j = 0; j < m; j++) {
5             matrix[i][j] = fill;
6         }
7     }
8 }
```

Листинг 3.2 – Функция нахождения расстояния Левенштейна с использованием матрицы

```
int lev_mtr(wstring &str1, wstring &str2, bool print ) {
export function nonrecursiveL(str1, str2) {
    let n = str1.length;
    let m = str2.length;
    let matrix = [];
    createMatrix(matrix, n + 1, m + 1, 0);
    for (let i = 0; i < n + 1; i++) {
        for (let j = 0; j < m + 1; j++) {
            if (i === 0 && j === 0) {
                matrix[i][j] = 0;
            }
            else if (i === 0) {
                matrix[i][j] = j;
            }
            else if (j === 0) {
                matrix[i][j] = i;
            }
            else {
                let diff = 1;
                if (str1[i - 1].localeCompare(str2[j - 1]) ===
                   0) {
                    diff = 0;
                }
                matrix[i][j] = Math.min(matrix[i][j - 1] + 1,
                   matrix[i - 1][j] + 1, matrix[i - 1][j - 1] +
                   diff);
            }
        }
    }

    return [matrix[n][m], matrix];
}
}
```

Листинг 3.3 – Функция нахождения расстояния Дамерау-Левенштейна с использованием матрицы

```
export function nonrecursiveDL(str1, str2) {
    let n = str1.length;
    let m = str2.length;
    let matrix = [];
    createMatrix(matrix, n + 1, m + 1, 0);
    for (let i = 0; i < n + 1; i++) {
        for (let j = 0; j < m + 1; j++) {
            if (i === 0 && j === 0) {
                matrix[i][j] = 0;
            }
            else if (i === 0) {
                matrix[i][j] = j;
            }
            else if (j === 0) {
                matrix[i][j] = i;
            }
            else {
                let diff = 1;
                if (str1[i - 1].localeCompare(str2[j - 1]) ===
                   0) {
                    diff = 0;
                }
                if (i > 1 && j > 1 && str1[i - 1] === str2[j -
                   2] && str1[i - 2] === str2[j - 1]) {
                    matrix[i][j] = Math.min(matrix[i][j - 1] +
                       1, matrix[i - 1][j] + 1, matrix[i - 1][j
                       - 1] + diff, matrix[i - 2][j - 2] + 1);
                }
                else {
                    matrix[i][j] = Math.min(matrix[i][j - 1] +
                       1, matrix[i - 1][j] + 1, matrix[i - 1][j
                       - 1] + diff);
                }
            }
        }
    }

    return [matrix[n][m], matrix];
}
```

Листинг 3.4 – Функция нахождения расстояния Дамерау-Левенштейна рекурсивно

```
export function recursiveDL(str1, str2) {
    let n = str1.length;
    let m = str2.length;
    if (n === 0 || m == 0) {
        return Math.abs(n - m);
    }
    let diff = 1;
    if (str1[n - 1].localeCompare(str2[m - 1]) === 0) {
        diff = 0;
    }
    let insertOp = recursiveDL(str1, str2.slice(0, m - 1)) + 1;
    let deleteOp = recursiveDL(str1.slice(0, n - 1), str2) + 1;
    let replaceOP = recursiveDL(str1.slice(0, n - 1),
       str2.slice(0, m - 1)) + diff;
    if (n > 1 && m > 1 && str1[n - 1] === str2[m - 2] && str1[n
      - 2] === str2[m - 1]) {
        return Math.min(insertOp, deleteOp, replaceOP,
           recursiveDL(str1.slice(0, n - 2), str2.slice(0, m -
           2)) + 1);
    }
    else {
        return Math.min(insertOp, deleteOp, replaceOP);
    }
}
```

Листинг 3.5 – Функция нахождения расстояния Дамерау-Левенштейна рекурсивно с кешированием

```
function recursiveDLCash (str1, str2, matrix){
    let n = str1.length;
    let m = str2.length;
    if(n === 0){
        matrix[n][m] = m;
        return m;
    }
    if(m === 0){
        matrix[n][m] = n;
        return n;
    }
    let diff = 1;
    if (str1[n - 1].localeCompare(str2[m - 1]) === 0) diff = 0;
    let insertOp = recursiveDLCash(str1, str2.slice(0, m - 1),
       matrix) + 1;
    let deleteOp = recursiveDLCash(str1.slice(0, n - 1), str2,
       matrix) + 1;
    let replaceOP = recursiveDLCash(str1.slice(0, n - 1),
       str2.slice(0, m - 1), matrix) + diff;
    if (n > 1 && m > 1 && str1[n - 1] === str2[m - 2] && str1[n
      - 2] === str2[m - 1]) {
        matrix[n][m] = Math.min(insertOp, deleteOp, replaceOP,
           recursiveDLCash(str1.slice(0, n - 2), str2.slice(0,
           m - 2), matrix) + 1);
    }
    else {
        matrix[n][m] = Math.min(insertOp, deleteOp, replaceOP);
    }
    return matrix[n][m];
}

export function DLCash(str1, str2) {
    let n = str1.length;
    let m = str2.length;
    let matrix = [];
    createMatrix(matrix, n + 1, m + 1, -1);
    let distance = recursiveDLCash(str1, str2, matrix);
    return [distance, matrix];
}
```

3.4 Функциональные тесты

В таблице 3.1 приведены функциональные тесты для алгоритмов вычисления расстояний Левенштейна и Дамерау—Левенштейна. Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты

Входные данные		**Расстояние и алгоритм**			
Строка 1	**Строка 2**	**Левенштейна**	**Дамерау-Левенштейна**		
		Итеративный	**Итеративный**	**Рекурсивный**	
				Без кеша	**С кешом**
a	b	1	1	1	1
ы	ы	0	0	0	0
кот	скат	2	2	2	2
увлечение	развлечения	4	4	4	4
алгоритм	лагоритмы	3	2	2	2
sator	rotas	4	4	4	4
игра	гиар	3	2	2	2

Вывод

Были реализованы алгоритмы поиска расстояния Левенштейна итеративно, а также поиска расстояния Дамерау–Левенштейна итеративно, рекурсивно и рекурсивно с кешированием. Проведено тестирование реализаций алгоритмов.

4 Исследовательская часть

В данном разделе будут приведены примеры работы программ, проведение исследования и сравнительный анализ алгоритмов на основе полученных данных.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры по времени:

— Процессор: 11th Gen Intel(R) Core(TM) i5-1135G7 2.42 GHz.

— Оперативная память: 16 ГБайт.

— Операционная система: Windows 11 Домашняя 64-разрядная система версии 22H2 [4].

При замерах времени ноутбук был включен в сеть электропитания и был нагружен только системными приложениями.

4.2 Демонстрация работы программы

Программа получает на вход 2 слова и выдает 4 расстояния, соответствующих алгоритмов поиска редакторского расстояния Левенштейна, итеративного Дамерау-Левенштейна, рекурсивного Дамерау-Левентейна, рекурсивного с кешированием Дамерау-Левенштейна.

На рисунках 4.1 – 4.2 представлена демонстрация работы программы.

Рисунок 4.1 – Демонстрация работы программы при поиске расстояний Левенштейна и Дамерау-Левенштейна (часть 1)

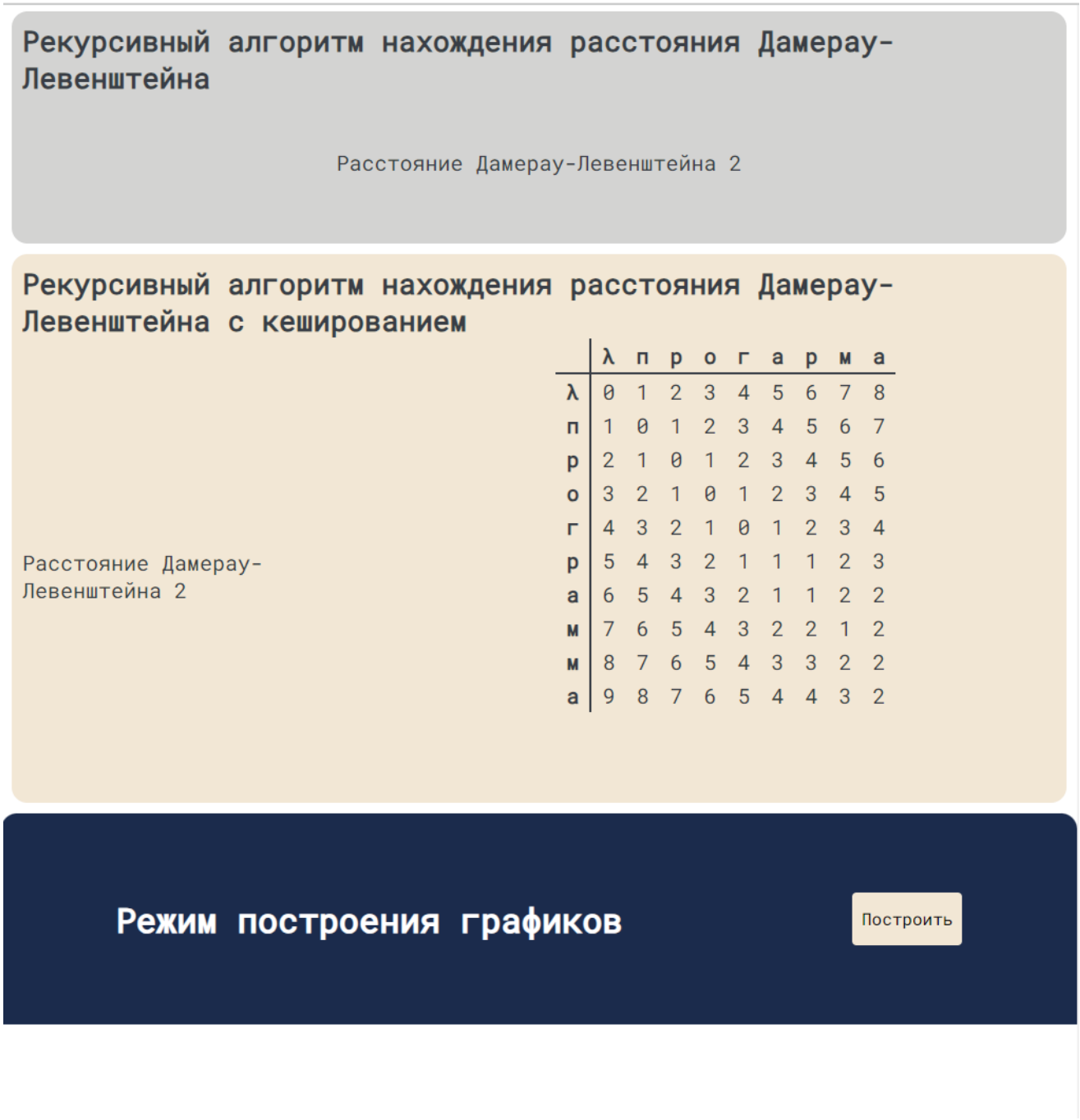

Рисунок 4.2 – Демонстрация работы программы при поиске расстояний Левенштейна и Дамерау-Левенштейна (часть 2)

4.3 Временные характеристики

Результаты тестирования приведены в таблице 4.1. Прочерк в таблице означает что тестирование для этого набора данных не выполнялось.

Замеры проводились на одинаковых длин строк от 1 до 200 с различным шагом.

Таблица 4.1 – Замер по времени для строк, размер которых от 1 до 200

Длина (символ)	**Время, нс**			
	Левенштейн	**Дамерау-Левенштейн**		
	Итеративный	**Итеративный**	**Рекурсивный**	
			Без кеша	**С кешом**
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
5	0	0	0	78.125
6	0	0	546.875	859.375
7	0	0	3359.375	3281.25
8	0	0	19375	26796.875
9	0	0	146328.125	114296.875
10	0	0	612578.125	692968.75
20	32.6	40.65	-	-
30	45.25	78.125	-	-
50	78.125	156.25	-	-
100	156.25	390.625	-	-
200	1015.625	1093.75	-	-

На рисунке 4.3 представлен график времени работы итеративных алгоритмов поиска расстояния Левенштейна и расстояния Дамерау-Левенштейна.

На рисунке 4.4 представлен график времени работы рекурсивных алгоритмов поиска расстояния Дамерау-Левенштейна с кешированием и без.

4.4 Характеристики по памяти

Алгоритмы Левенштейна и Дамерау-Левенштейна имеют схожие требования к использованию памяти, поэтому мы рассмотрим разницу между рекурсивной и матричной реализациями одного из этих алгоритмов.

Рисунок 4.3 – Сравнение по времени нерекурсивных реализаций алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна

В случае рекурсивной реализации максимальная глубина стека вызовов равна сумме длин входных строк. Каждый вызов функции также потребляет дополнительные 6 переменных типа *number*, и строка, передаваемая в аргументах функции, может быть обрезана на один символ с каждым вызовом. Таким образом, максимальное потребление памяти определяется следующим образом:

$$(Len(S_1)+Len(S_2))\cdot 8\cdot Size(number)+\left(\frac{(n+1)\cdot n}{2}+\frac{(m+1)\cdot m}{2}\right)\cdot Size(char), \tag{4.1}$$

где S_1 и S_2 - строки, Size - функция, возвращающая размер аргумента; Len - функция, возвращающая длину строки, char - символьный тип, number - число двойной точности.

С другой стороны, потребление памяти при использовании итеративной реализации теоретически оценивается как:

$$(Len(S_1)+1)\cdot(Len(S_2)+1)\cdot Size(number)+3\cdot Size(number)+Size(S_1)+Size(S_2). \tag{4.2}$$

С точки зрения потребления памяти, итеративные алгоритмы уступают рекурсивным: максимальный объем используемой памяти в итеративных алгоритмах растет как произведение длин строк, в то время как у рекурсивного алгоритма он увеличивается как сумма длин строк.

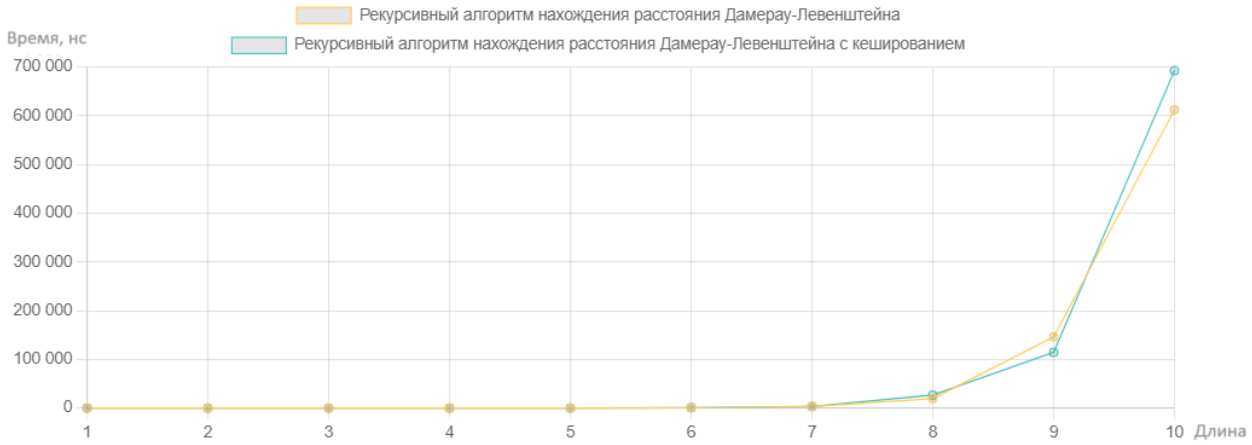

Рисунок 4.4 – Сравнение по времени рекурсивных алгоритмов поиска расстояния Дамерау-Левенштейна

4.5 Вывод

В данном разделе были сравнены алгоритмы по памяти и по времени. Рекурсивный алгоритм Дамерау – Левенштейна работает дольше итеративных реализаций — время этого алгоритма увеличивается в геометрической прогрессии с ростом размера строк. Рекурсивный алгоритм с кешированием превосходит простой рекурсивный алгоритм по времени. По расходу памяти все реализации проигрывают рекурсивной за счет большого количества выделенной памяти под матрицу расстояний. Таким образом, самым эффективным по памяти является рекурсивный алгоритм, а самый эффективным по времени - итеративный алгоритм (исходя из сделанных тестов).

Заключение

Было экспериментально подтверждено различие во временной эффективности рекурсивной и нерекурсивной реализаций выбранных алгоритмов нахождения расстояния между строками при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализаций на различных длинах строк. В результате исследований можно сделать вывод о том, что матричная реализация данных алгоритмов заметно выигрывает по времени при росте длин строк, но проигрывает по количеству затрачиваемой памяти.

В ходе выполнения данной лабораторной работы были решены следующие задачи:

1) Описаны алгоритмы вычисления расстояний Левенштейна и Дамерау-Левенштейна.

2) Разработано программное обеспечение, включающее в себя следующие алгоритмы:

   — нерекурсивный алгоритм для вычисления расстояния Левенштейна;

   — нерекурсивный алгоритм для вычисления расстояния Дамерау-Левенштейна;

   — рекурсивный алгоритм для вычисления расстояния Дамерау-Левенштейна;

   — рекурсивный с кешированием алгоритм для вычисления расстояния Дамерау-Левенштейна.

3) Выбраны инструменты для замера процессорного времени выполнения реализаций алгоритмов.

4) Проведен анализ затрат по времени и памяти для каждой реализации.

Список использованных источников

1 И. Левенштейн В. Двоичные коды с исправлением выпадений, вставок и замещений символов. — М.: Издательство «Наука», Доклады АН СССР, 1965. Т. 163.

2 Документация по JavaScript [Электронный ресурс]. — Режим доступа: `https://262.ecma-international.org/13.0/` (дата обращения: 17.09.2023).

3 Node.js function process.cpuUsage([previousValue]) [Электронный ресурс]. — Режим доступа: `https://nodejs.org/api/process.html#processcpuusagepreviousvalue` (дата обращения: 17.09.2023).

4 Windows 11 [Электронный ресурс]. — Режим доступа: `https://www.microsoft.com/ru-ru/software-download/windows11` (дата обращения: 18.09.2023).