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»

ОТЧЕТ

по Лабораторной работе №1

по курсу «Анализы Алгоритмов»

на тему: «Динамическое программирование»

Студент группы ИУ7-56Б ____________ Мамврийский И. С.
(Подпись, дата) (Фамилия И.О.)

Преподаватель ____________ Волкова Л. Л.
(Подпись, дата) (Фамилия И.О.)

Преподаватель ____________ Строганов Ю. В..
(Подпись, дата) (Фамилия И.О.)

Москва — 2023 г.

Содержание

Введение

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

Оно 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.

Цели данной лабораторной работы:

1) Описать алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна.

2) Создать программное обеспечение, реализующее следующие алгоритмы:

    — нерекурсивный алгоритм поиска расстояния Левенштейна;
    — нерекурсивный алгоритм поиска расстояния Дамерау-Левенштейна;
    — рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна;
    — рекурсивный с кешированием алгоритм поиска расстояния Дамерау-Левенштейна.

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

4) Провести анализ затрат реализаций алгоритмов по времени и по памяти, определить влияющие на них характеристики.

1 Аналитическая часть

Расстояние Левенштейна(редакционное расстояние, дистанция редактирования) - минимальное количество редакционных операций вставки, удаления и замены, необходимых для превращения одной строки в другую.

Цены операций могут зависеть от вида операций и/или от участвующих в ней символов:

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.

Нам необходимо найти последовательность, минимализирующую суммарную цену. Расстояние Левенштейна является частным случаем решения данной зхадачи при:

- $w(a, a) = 0$;
- $w(a, b) = 1$, $a \neq b$, в противном случае замена не происходит;
- $w(\lambda, b) = 1$;
- $w(a, \lambda) = 1$.

Введем понятие M, которое будет обозначать совпадение, то есть $w(a, a) = 0$.

1.1 Матричный алгоритм нахождения расстояния Левенштейна

Расстояние между двумя строками a и b может быть вычислено по формуле 1.1, с использованием матрицы рамзером (N + 1) * (M + 1) для сохранения соответствующих промежуточных значений. Данный алгоритм представляет собой построчное заполнение матрицы $A_{|a|,|b|}$ значениями $D(i, j)$, где [i, j] - значение ячейки. Первая ячейчка заполняется 0, остальные в соответствии с формулой:

$$D(i,j) = \begin{cases} 0 & \text{если } i = 0, j = 0 \\ i & \text{если } j = 0, i > 0 \\ j & \text{если } i = 0, j > 0 \\ \min\{ & \\ \qquad D(i, j-1) + 1 & \\ \qquad D(i-1, j) + 1 & \text{если } i > 0, j > 0 \\ \qquad D(i-1, j-1) + m(a[i], b[j]) & (1.2) \\ \} & \end{cases} \tag{1.1}$$

а функция 1.2 определена как:

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

Функция D составлена по следущему принципу, где a[i] - i-ый символ строки a, b[j] - j-ый символ строки b, функция D(i, j) определена как:

1) Из перевода пустой строки в пустую требуется 0 операций.

2) Из перевода пустой строки в строку a треубется $|a|$ операций.

3) Из перевода строки в пустую строку a треубется $|a|$ операций.

4) Для перевода из строки a в строку b требуется выполнить несколько операций(удаления, добавления, замены). Пусть a' и b' строки a и b без последнего символа, поэтому цена преобрахзования строки a в строку b может выглядеть следующшим образом:

    a) Сумма цены преобразования строки a в b и цена проведения операции удаления, которая необходима для преобразваония a' в a.

    b) Сумма цены преобразования строки a в b и цены проведения операции вставки, которая необходима для преобразваония b' в b.

c) Сумма цены преобразования из a' в b' и цена операции замены, предполагая, что a и b оканчиваются на разные символы.

d) Сумма цены преобразоания из a' в b', предполагая, что a и b оканчиваются на один и тот же символ.

Минимальная цена преобразования - минимальное значение приведенных операций.

1.2 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна может быть найдено по формуле 1.3, которая задана как:

$$d_{a,b}(i,j) = \begin{cases} \max(i,j), & \text{если } \min(i,j)=0, \\ \min\{ & \\ \quad d_{a,b}(i,j-1)+1, & \\ \quad d_{a,b}(i-1,j)+1, & \\ \quad d_{a,b}(i-1,j-1)+m(a[i],b[j]), & \text{иначе} \\ \quad \left[\begin{array}{ll} d_{a,b}(i-2,j-2)+1, & \text{если } i,j>1; \\ & a[i]=b[j-1]; \\ & b[j]=a[i-1] \\ \infty, & \text{иначе} \end{array} \right. & \\ \} & \end{cases}, \tag{1.3}$$

Данная формула выводится по тем же соображениям, что и формула 1.1, однако, в этой формуле добавляется еще одного условие для случая, когда обе строки не пустые.

Сумма цен преобразования из a" в b" и операции перестановки, предполагая, что длины a" и b" больше 1 и последние два символа a", если их поменять местами, совпадут с последними двумя символами b".

1.3 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна с кешированием

Рекурсивная реализацич алгоритм Домера-Левенштейна малоэффективна при больших M и N, так как некоторые значения будут вычислены повторно. Для оптимизации данного алгоритма можно использовать кеш в виде матрицы для сохранения промежуточных знаяений значений. В таком случае алгоритм представляет собой рекурсивное заполнение матрицы $A_{|a|,|b|}$. промежуточными значениями D(i, j).

1.4 Нерекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна

Рекурсивная реализация алгоритма Левенштейна с кешированием малоэффективна по времени при больших M и N. Для быстроты действия можно использовать нереукрсивный алгоритм. Он представляет собой итерационную реализацию заполнения матрицы промежуточными значение D(i, j).

В качетсве структуры даннвх для хранения можно использовать матрицу размером:

$$(N+1) \times (M+1), \tag{1.4}$$

Значение в ячейке [i, j] равно зачению $D(S1[1...i], S2[1...j])$. Первый элемент заполняется нулем, остальные в соответствии с формулой (1.3).

1.5 Вывод

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

2 Конструкторская часть

2.1 Схемы алгоритмов

В данной части будут рассмотрены схемы алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна. На рисунках 2.1-2.4 представлены данные алгоритмы.

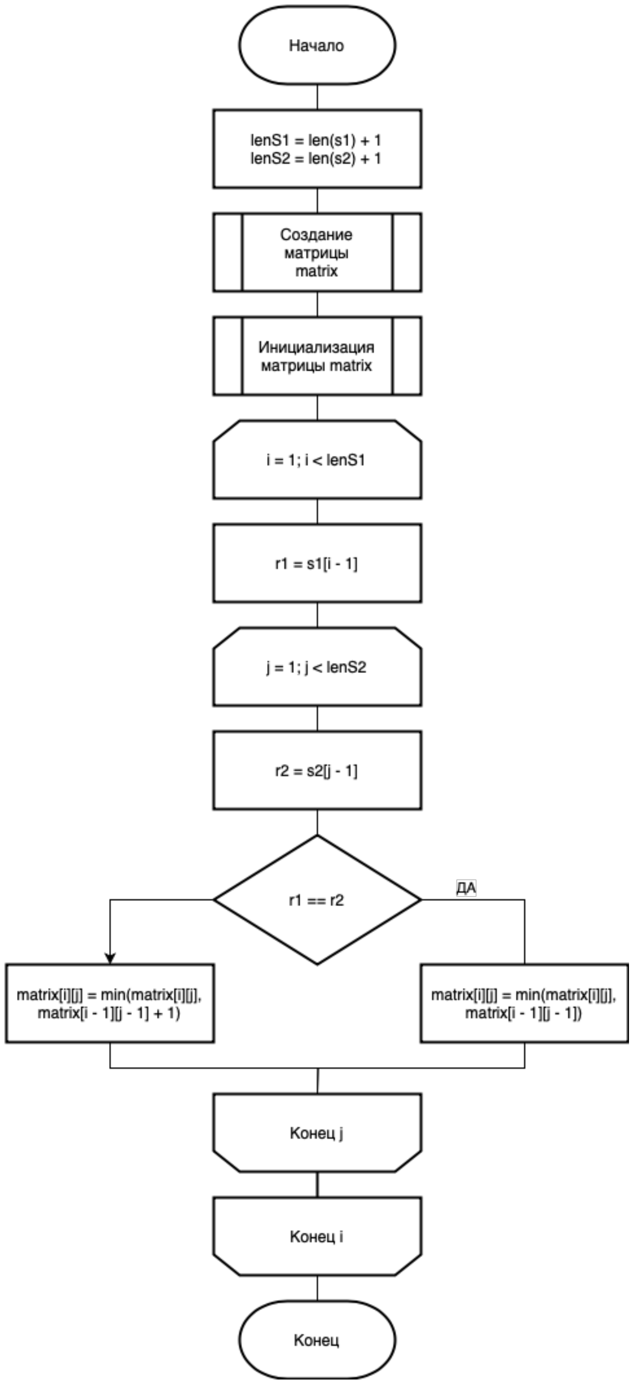

Рисунок 2.1 – Схема нерекурсивного алгоритма нахождения расстояния Левенштейна

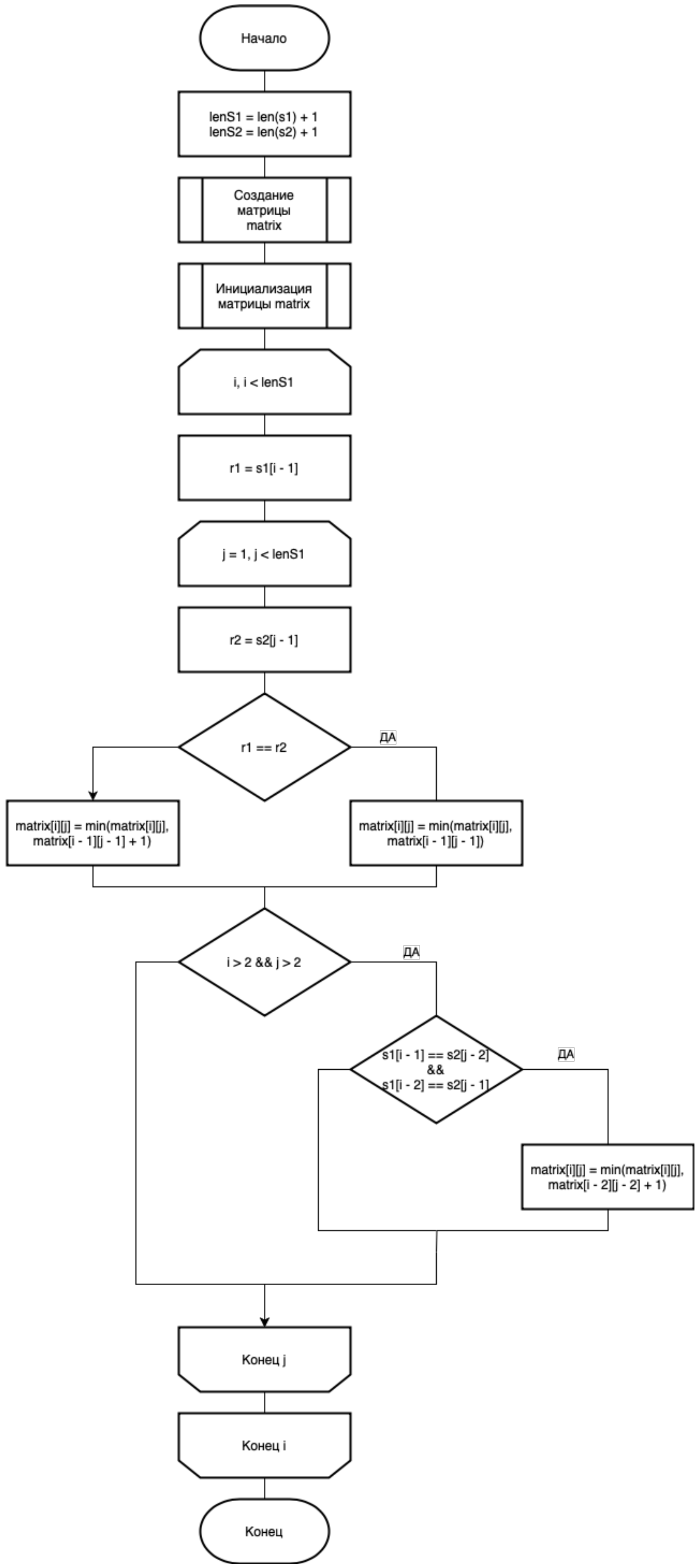

Рисунок 2.2 – Схема нерекурсивного алгоритма нахождения расстояния Левенштейна-Дамерау

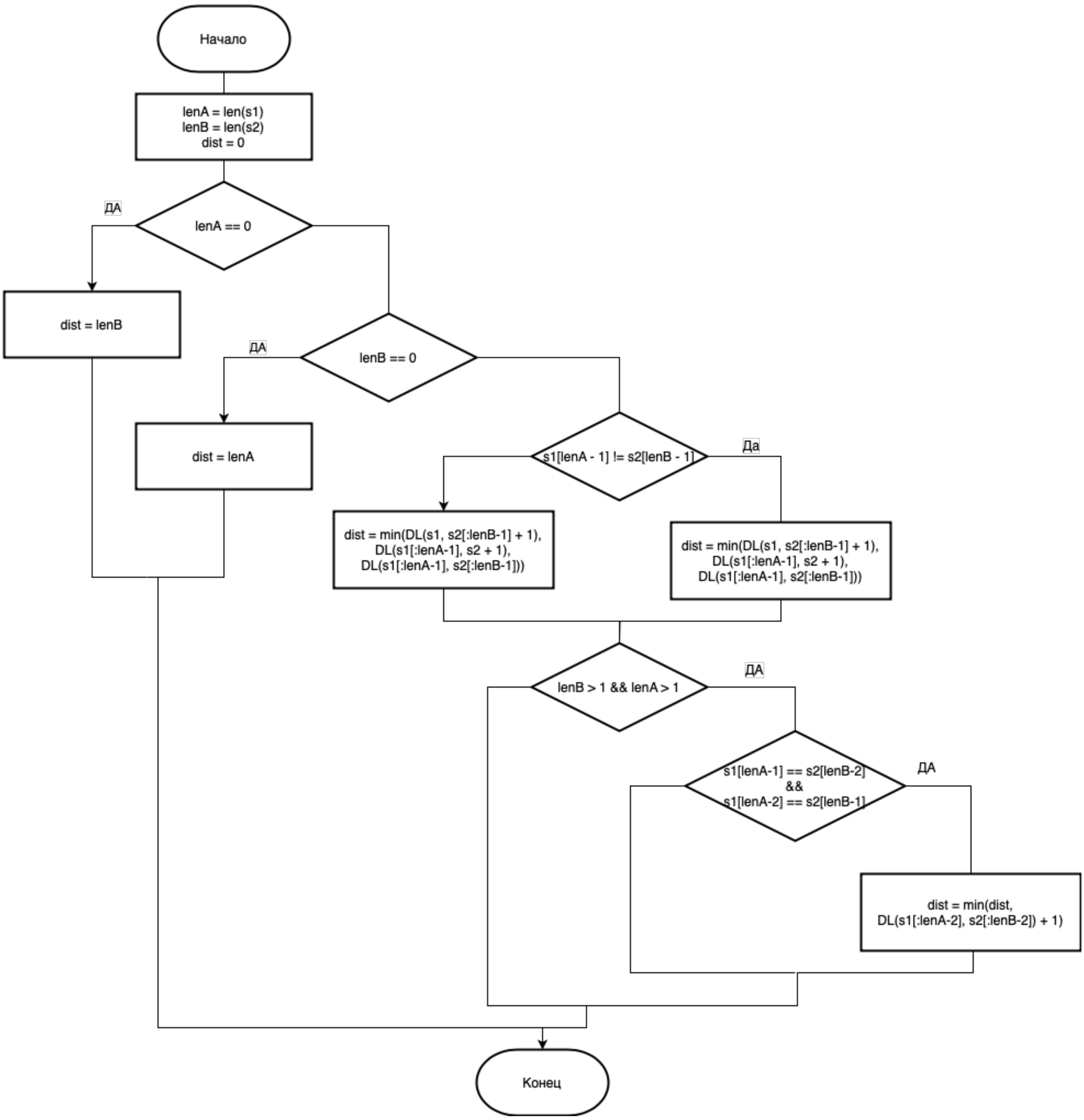

Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Левенштейна-Дамерау

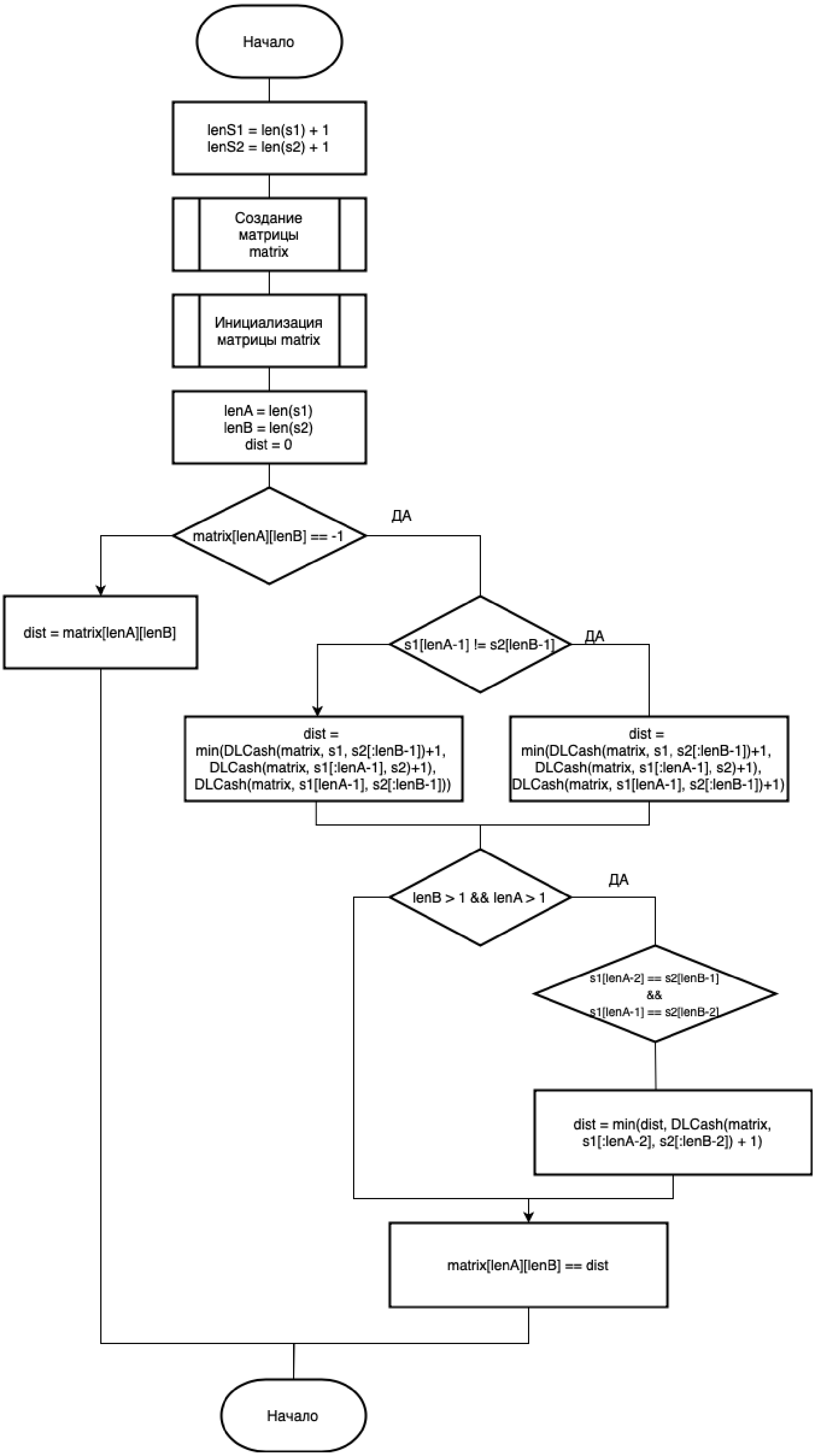

Рисунок 2.4 – Схема рекурсивного алгоритма нахождения расстояния Левенштейна-Дамерау с кешированием

2.2 Вывод

На основе теоретических даннных, полученных в аналитическом разделе были построены схемы исселедуемых алгоритмов.

3 Технологический раздел

3.1 Требования к ПО

Требования к вводу:

1) На вход подаются две строки в любой раскладке (в том числе и пустые);

2) ПО должно выводить полученное расстояние;

3.2 Средства реализации

Для реализации программы нахождения расстояние Левенштейна я выбрал язык программирования Golang. Данный выбор обусловлен моим желанием расширить свои знания в области применения данного язкыа программирования.

3.3 Реализация алгоритмов

В следующих листингах приведена реализация алгоритмов нахождения расстония Левенштейна и Дамерау-Левенштейна.

Листинг 3.1 – Функция нахождения расстояния Левенштейна нерекурсивным способом

```
func LM(matrix [][]int, a, b int, s1, s2 []rune) int {
    for i := 1; i < a; i++ {
        r2 := s2[i - 1]
        for j := 1; j < b; j++ {
            r1 := s1[j - 1]
            if r1 != r2 {
                matrix[i][j] = min(matrix[i][j - 1] + 1,
                   matrix[i - 1][j] + 1, matrix[i - 1][j - 1] +
                   1)
            } else {
                matrix[i][j] = min(matrix[i][j - 1] + 1,
                   matrix[i - 1][j] + 1, matrix[i - 1][j - 1])
            }
        }
    }

    return matrix[a - 1][b - 1]
}
```

Листинг 3.2 – Функция нахождения расстояния Левенштейна-Дамерау с помощью рекурсии

```
func DL(s1, s2 []rune) int {
    lenB := len(s1)
    lenA := len(s2)

    dist := 0

    if lenA == 0 {
        dist = lenB
    } else if lenB == 0 {
        dist = lenA
    } else {
        k := 0
```

```
        if s1[lenB - 1] != s2[lenA - 1] {
            k = 1
        }

        dist = min(DL(s1, s2[ : lenA - 1]) + 1, DL(s1[ : lenB -
          1], s2) + 1, DL(s1[ : lenB - 1], s2[ : lenA - 1]) +
          k)

        if lenB > 1 && lenA > 1 && s1[lenB - 1] == s2[lenA - 2]
          && s1[lenB - 2] == s2[lenA - 1] {
            dist = min(dist, DL(s1[ : lenB - 2], s2[ : lenA -
              2]) + 1)
        }
    }

    return dist
}
```

Листинг 3.3 – Функция нахождения расстояния Левенштейна-Дамерау с помощью рекурсии и кеша

```
func DLCash(matrix [][]int, s1, s2 []rune) int {
    lenB := len(s1)
    lenA := len(s2)

    dist := 0

    if matrix[lenA][lenB] == -1 {
        k := 0
        if s1[lenB - 1] != s2[lenA - 1] {
            k = 1
        }

        dist = min(DLCash(matrix, s1, s2[ : lenA - 1]) + 1,
          DLCash(matrix, s1[ : lenB - 1], s2) + 1,
          DLCash(matrix, s1[ : lenB - 1], s2[ : lenA - 1]) + k)

        if lenB > 1 && lenA > 1 && s1[lenB - 1] == s2[lenA - 2]
          && s1[lenB - 2] == s2[lenA - 1] {
            dist = min(dist, DLCash(matrix, s1[ : lenB - 2],
              s2[ : lenA - 2]) + 1)
        }
```

```

        matrix[lenA][lenB] = dist
    } else {
        dist = matrix[lenA][lenB]
    }

    return dist
}
```

Листинг 3.4 – Функция нахождения расстояния Дамерау-Левенштейна нерекурсивным способом

```
func DLM(matrix [][]int, a, b int, s1, s2 []rune) int {
    for i := 1; i < a; i++ {
        for j := 1; j < b; j++ {
            if s1[j - 1] != s2[i - 1] {
                matrix[i][j] = min(matrix[i][j - 1] + 1,
                    matrix[i - 1][j] + 1, matrix[i - 1][j - 1] +
                    1)
            } else {
                matrix[i][j] = min(matrix[i][j - 1] + 1,
                    matrix[i - 1][j] + 1, matrix[i - 1][j - 1])
            }

            if i > 2 && j > 2 && s1[j - 1] == s2[i - 2] && s1[j
              - 2] == s2[i - 1] {
                matrix[i][j] = min(matrix[i][j], matrix[i -
                    2][j - 2] + 1)
            }
        }
    }

    return matrix[a - 1][b - 1]
}
```

3.4 Тестирование

Входные данные		**Расстояние и алгоритм**			
		Левенштейна	**Дамерау-Левенштейна**		
Строка 1	**Строка 2**	**Итеративный**	**Итеративный**	**Рекурсивный**	
				Без кеша	**С кешом**
a	d	1	1	1	1
a	a	0	0	0	0
переговоры	перегрел	5	5	5	5
cat	cats	1	1	1	1
кот	кто	2	1	1	1
12345	54321	4	4	4	4

Таблица 3.1 – Тестовые данные

Вывод

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

4 Исследовательская часть

4.1 Прмиер работы

Демонстрация работы программы приведена на рисунке 4.1

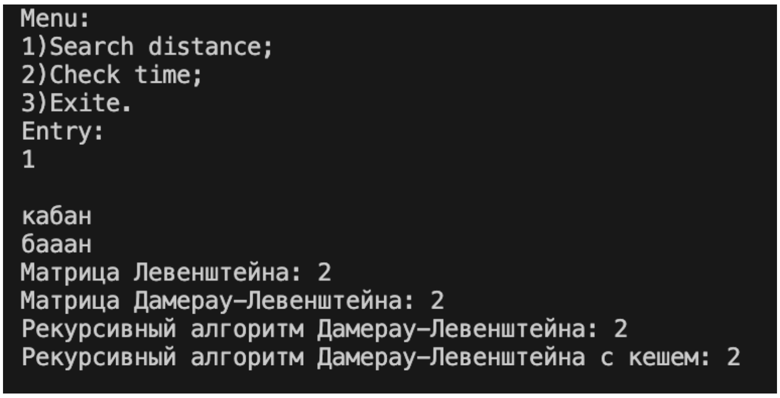

Рисунок 4.1 – Демонстрация работы программы при поиске расстояний Левенштейна и Дамерау-Левенштейна

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры по времени, представлены далее.

— Процессор:2 GHz 4-ядерный процессор Intel Core i5.

— Оперативная память: 16 ГБайт.

— Операционная система: macOS Venura 13.5.2.

4.3 Время выполнения алгоритмов

Результаты эксперимента замеров по времени приведены в таблице 4.2. В данной таблице присутствуют поля с «-». Это обусловлено тем, что для рекурсивной реализации алгоритма достаточно приведенных замеров для построения графика. При длине строки большей 10 замер времени для рекурсивного алгоритма будет достаточно долгим. Замеры проводились на одинаковых длин строк от 1 до 1000 с различным шагом.

LENGHT	MATRXL	MATRIXDL	RDL	RCDL
1	7000	2000	2000	1000
2	24000	2000	2000	1000
3	10000	2000	2000	2000
4	18000	2000	2000	4000
5	17000	2000	2000	21000
6	9000	1000	3000	105000
7	2000	2000	4000	617000
8	24000	2000	4000	3098000
9	11000	3000	9000	12439000
10	18000	2000	5000	38905000
30	11000	15000	22000	-
50	30000	22000	86000	-
70	49000	39000	111000	-
90	81000	69000	219000	-
110	98000	101000	301000	-
130	139000	122000	390000	-
150	199000	187000	520000	-
170	218000	231000	698000	-
190	283000	1065000	866000	-
210	197000	190000	891000	-
230	230000	226000	1012000	-
250	765000	394000	1929000	-
270	301000	299000	2388000	-
290	945000	346000	1782000	-
310	485000	985000	1876000	-
330	450000	422000	2733000	-
350	463000	473000	2910000	-
370	493000	899000	2734000	-
390	1160000	578000	3398000	-
410	696000	1044000	3441000	-
430	1207000	868000	5289000	-
450	1264000	1284000	5057000	-
470	1513000	891000	5030000	-
490	1339000	1330000	5341000	-
510	1370000	1376000	6232000	-
530	2430000	1695000	7057000	-
550	1546000	1606000	6566000	-
570	1788000	1941000	8424000	-
590	1736000	1809000	7529000	-
610	2138000	2266000	8548000	-
630	1980000	3092000	9131000	-
650	2234000	2025000	9546000	-
670	2192000	2204000	10795000	-
690	2389000	3008000	12776000	-
710	2359000	2861000	13873000	-
730	2624000	3440000	13781000	-
750	4280000	2945000	13969000	-
770	3455000	3671000	15509000	-
790	3592000	3835000	16446000	-
810	3607000	3594000	16551000	-
830	4081000	3990000	19841000	-
850	3453000	3895000	19560000	-
870	5150000	4325000	20424000	-
890	5638000	4441000	21397000	-
910	4712000	5422000	22408000	-
930	5182000	5428000	23051000	-
950	5175000	6191000	24538000	-
970	5785000	5422000	24033000	-
990	5651000	4880000	23959000	-

Рисунок 4.2 – Демонстрация работы программы при поиске расстояний Левенштейна и Дамерау-Левенштейна

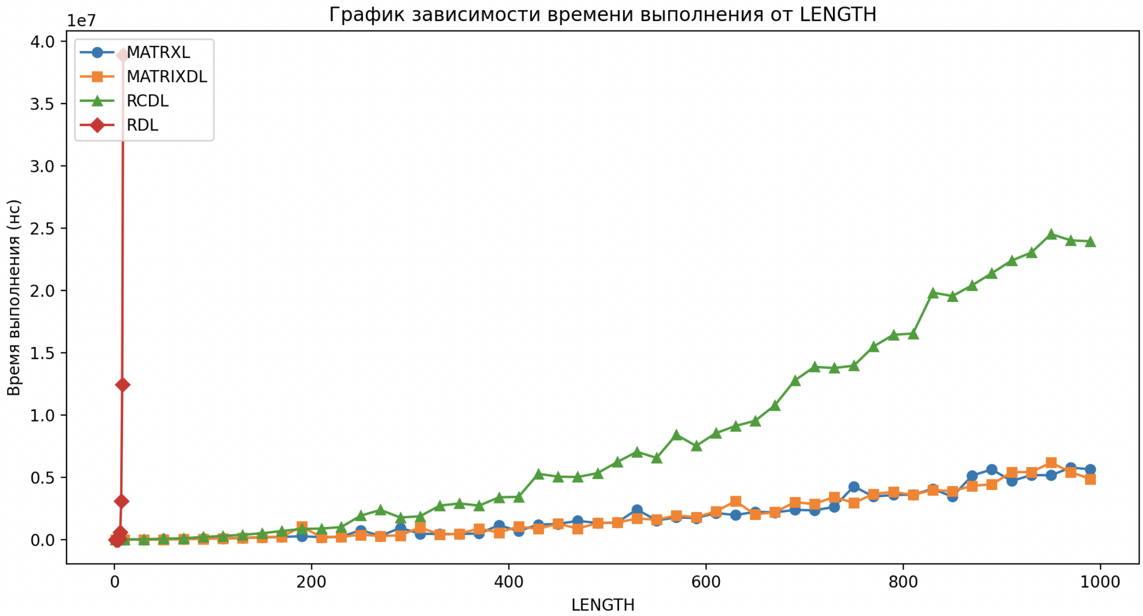

Рисунок 4.3 – График работы программы при поиске расстояний Левенштейна и Дамерау-Левенштейна

При длинах строк менее 200 символов разница по времени между итеративными реализациями и рекурсивной с кешем незначительна, однако при увеличении длины строки алгоритм рекурсивного поиска расстояния Дамерау-Левенштейна с кешем оказывается медленнее вплоть до 4 раз(при длинах строк равных 990).

Если рассматривать итерационные алгоритмы, то алгоритм Левенштейна работает немного быстрее, чем алгоритм Дамерау-Левенштейна, так как во втором есть дополнительная проверка.

Алгоритмы нахождения расстояния Левенштейна и Дамерау — Левенштейна не отличаются друг от друга с точки зрения использования памяти.

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк. Поэтому, максимальный расход памяти равен:

4.4 Использование памяти

$$(\mathcal{S}(STR_1) + \mathcal{S}(STR_2)) \cdot (2 \cdot \mathcal{S}(\text{string}) + 3 \cdot \mathcal{S}(\text{integer})), \tag{4.1}$$

где $\mathcal{S}$ — оператор вычисления размера, STR_1, STR_2 — строки, string — строковый тип,
integer — целочисленный тип.

Использование памяти при итеративной реализации теоретически равно:

$$(\mathcal{S}(STR_1)+1)\cdot(\mathcal{S}(STR_2)+1)\cdot\mathcal{S}(\text{integer})+5\cdot\mathcal{S}(\text{integer})+2\cdot\mathcal{S}(\text{string}). \tag{4.2}$$

4.5 Вывод

В данном разделе было произведено сравнение количества затраченного времени и памяти алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна. Наименее затратным по времени оказался итеративный алгоритм нахождения расстояния Левенштейна.

Приведенные характеристики показывают, что рекурсивная реализация алгоритма во много раз проигрывает по времени. В связи с этим, рекурсивные алгоритмы следует использовать лишь для малых размерностей строк (<= 10 символов).

Так как во время печати очень часто возникают ошибки связанные с транспозицией букв, алгоритм поиска расстояния Дамерау-Левенштейна является наиболее предпочтительным, не смотря на то, что он незначительно проигрывает по времени и памяти алгоритму Левенштейна.

5 Заключение

В ходе проделанной работы был изучен метод динамического программирования на материале реализации алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна. Также были изучены алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками и 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 процессорного времени выполнения реализации на варьирующихся длинах строк. Рекурсивная реализация алгоритма Левенштейна проигрывает нерекурсивной по времени исполнения в несколько десятков раз. Так же стоит отметить, что итеративный алгоритм Левештейна выполняется немного быстрее, чем итеративный алгоритм Дамерау - Левенштейна, но в целом алгоритмы выполняются за примерно одинаковое время.

Теоретически было рассчитано использования памяти в каждой из реализаций алгоритмов нахождения расстояния Левенштейна и Дамерау - Левенштейна. Рекурсивный алгоритм с мемоизацией в несколько десятков раз больше памяти, чем итеративная реализация алгоритма нахождения расстояния Левенштейна, из-за рекурсивного копирования вспомогательной матрицы.