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 "Анализ алгоритмов"

Тема Расстояния Левенштейна и Дамерау-Левенштейна

Студент Золотухин А.В.

Группа ИУ7-54Б

Оценка (баллы)

Преподаватели Волкова Л.Л., Строганов Ю.В.

Москва — 2022 г.

Оглавление

Введение

Целью данной лабораторной работы является изучение, реализация и исследование алгоритмов нахождения расстояний Левенштейна и Дамерау–Левенштейна.

Расстояние Левенштейна (редакционное расстояние, дистанция редактирования) — метрика, измеряющая разность между двумя последовательностями символов. Она определяется как минимальное количество односимвольных операций (вставки, удаления, замены), необходимых для превращения одной строки в другую. Широко используется в теории информации и компьютерной лингвистике.

Расстояние Левенштейна **Расстояние Левенштейна [1]** и его обобщения активно применяются для:

- исправления ошибок в слове (в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи);
- сравнения текстовых файлов утилитой `diff` и ей подобными (здесь роль «символов» играют строки, а роль «строк» — файлы);
- сравнения генов, хромосом и белков в биоинформатике.

Расстояние Дамерау — Левенштейна (названо в честь учёных Фредерика Дамерау и Владимира 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. Одной из самых частых ошибок в слове является перестановка соседних символов, поэтому Дамерау модифицировал расстояние Левенштейна, добавив операцию транспозиции (перестановки) символов.

Задачами данной лабораторной являются:

1. изучение алгоритмов Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;
2. применение метода динамического программирования для реализации алгоритмов;
3. получение практических навыков реализации алгоритмов Левенштейна и Дамерау — Левенштейна;
4. сравнительный анализ алгоритмов определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);
5. экспериментальное подтверждение различий во временно́й эффективности алгоритмов определения расстояния между строками при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк;
6. описание и обоснование полученных результатов в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

1 Аналитическая часть

В данном разделе будут представлены описания алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна и их практическое применение.

1.1 Расстояние Левенштейна

Расстояние Левенштейна [1] между двумя строками — это минимальное количество операций вставки, удаления и замены, необходимых для превращения одной строки в другую.

Каждая операция имеет свою стоимость. Редакционным предписанием называется последовательность действий, необходимых для получения из первой строки вторую, и минимизирующую суммарную цену. Суммарная цена есть искомое расстояние Левенштейна.

Введем следующие обозначения:

- D (delete) — удаление ($w(a, \lambda) = 1$);
- I (insert) — вставка ($w(\lambda, b) = 1$);
- R (replace) — замена ($w(a, b) = 1,\ a \neq b$);
- M (match) - совпадение ($w(a, a) = 0$).

Пусть S_1 и S_2 — две строки (длиной M и N соответственно) над некоторым алфавитом, тогда расстояние Левенштейна можно подсчитать по следующей рекуррентной формуле:

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

Функция m определена как:

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

Минимальной ценой преобразования будет минимальное значение приведенных вариантов.

1.2 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна [2] - это мера разницы двух строк символов, определяемая как минимальное количество операций вставки, удаления, замены и транспозиции, необходимых для перевода одной строки в другую. Является модификацией расстояния Левенштейна: к операциям вставки, удаления и замены символов, определённых в расстоянии Левенштейна добавлена операция транспозиции (перестановки) символов.

Расстояние Дамерау-Левенштейна для строк a, b может быть найдено по формуле 1.3, которая задана как

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

Формула выводится по тем же соображениям, что и формула (1.1).

1.3 Матричный алгоритм нахождения расстояния Дамерау-Левенштейна

При больших i, j реализация формулы 1.3 может быть очень затратна по времени исполнения, так как множество промежуточных значения $D(i,j)$ вычисляются не один раз . Для оптимизации нахождения можно использовать матрицу для хранения соответствующих промежуточных значений.

Матрица размером $(length(S1)+1)$ x $((length(S2)+1)$, где $length(S)$ — длина строки S. Значение в ячейке $[i,j]$ равно значению $D(S1[1...i], S2[1...j])$. Первая строка и первый столбец тривиальны.

Всю таблицу (за исключением первого столбца и первой строки) запол-

няем в соответствии с формулой 1.4.

$$A[i][j] = min \begin{cases} A[i-1][j] + 1 \\ A[i][j-1] + 1 \\ A[i-1][j-1] + m(S1[i], S2[j]) \\ A[i-2][j-2] + xch(S1[i], S2[j]) \end{cases} \tag{1.4}$$

$$xch(S1[i], S2[j]) = \begin{cases} 1 & \text{, если } i > 1, j > 1, S1[i-1] = S2[j], \\ & S1[i] = S2[j-1] \\ \infty & \text{, иначе.} \end{cases} \tag{1.5}$$

В результате расстоянием Левенштейна будет ячейка матрицы с индексами $i = length(S1)$ и $j = length(S2)$.

1.4 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна

Рекурсивный алгоритм реализует формулу 1.3. Функция D составлена таким образом, что для перевода из строки a в строку b требуется выполнить последовательно некоторое количество операций (удаление, вставка, замена) в некоторой последовательности. Полагая, что a', b' — строки a и b без последнего символа соответственно, цена преобразования из строки a в строку b может быть выражена как один из следующих случаев:

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

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

3) сумма цены преобразования из a' в b' и операции замены, если a и b оканчиваются на разные символы;

4) цена преобразования из a' в b', если a и b оканчиваются на один и тот же символ.

5) сумма цены преобразования строки a' в b и цены проведения операции обмена, если a и b последний символ a равен предпоследн

1.5 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна с использованием кеша

Рекурсивный алгоритм заполнения можно оптимизировать по времени выполнения с использованием кеша. В качестве кеша используется матрица. В случае, если рекурсивный алгоритм выполняет расчет для данных, которые еще не были обработаны (в ячейке находится бесконечность), результат нахождения расстояния заносится в матрицу. В случае, если ячейка не равна бесконечности, расстояние не находится и алгоритм переходит к следующему шагу.

Вывод

Формулы Левенштейна и Дамерау-Левенштейна для расчета расстояния между строками задаются рекуррентно, а следовательно, алгоритмы могут быть реализованы рекурсивно или итерационно.

2 Конструкторская часть

В этом разделе будут приведены требования к вводу и программе, а также схемы алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна.

2.1 Требования к вводу

Выделены следующие требования.

1. На вход подаются две строки.
2. Буквы верхнего и нижнего регистров считаются различными.

2.2 Требования к программе

Выделены следующие требования.

1. Две пустые строки — корректный ввод, программа не должна аварийно завершаться.
2. На выход программа должна вывести число — расстояние Левенштейна (Дамерау-Левенштейна), матрицу, если она используется алгоритмом.

2.3 Разработка алгоритма нахождения расстояния Левенштейна

На рисунке 2.1 приведена схема алгоритма нахождения расстояния Левенштейна с заполнением матрицы.

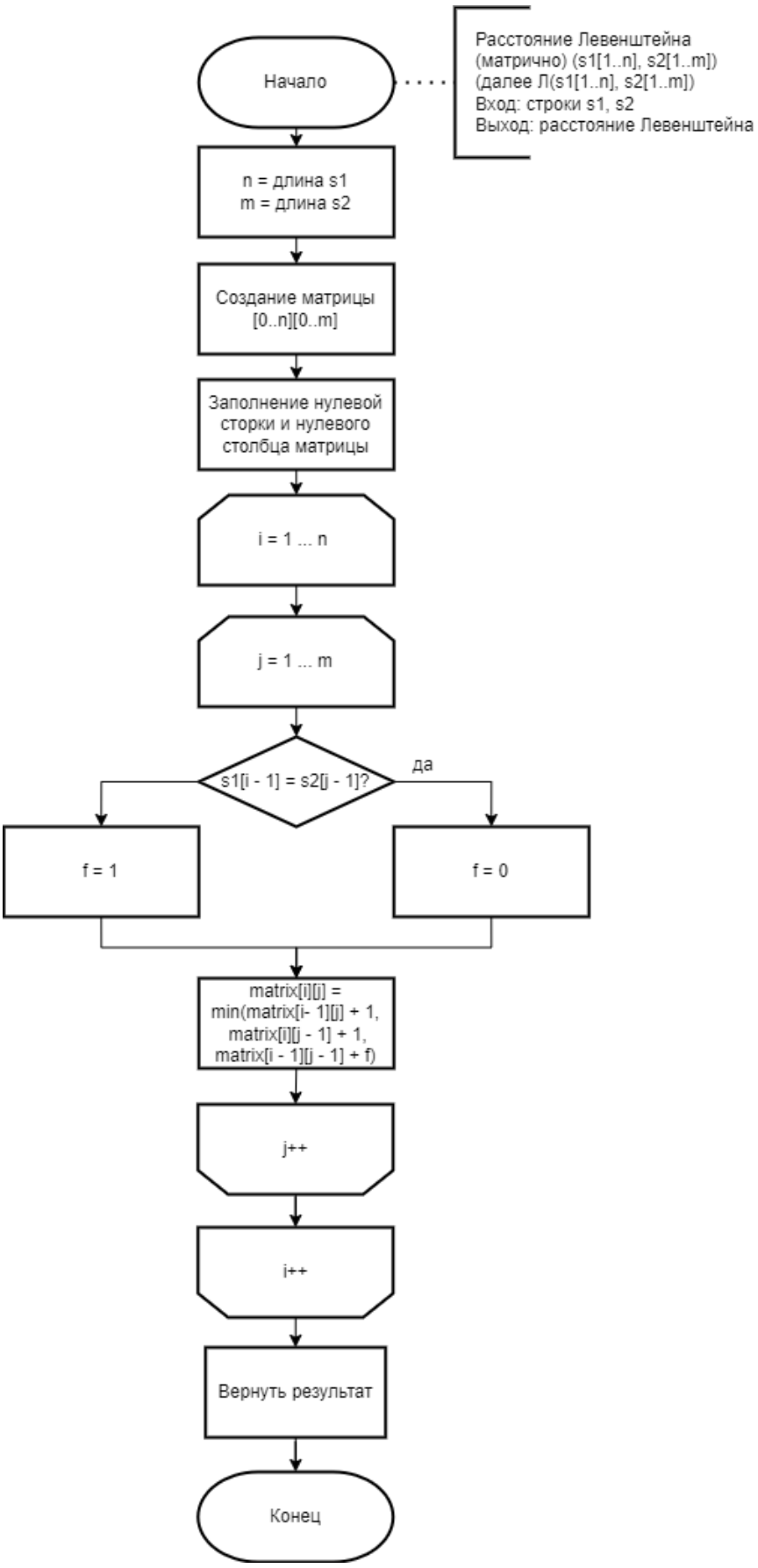

Рисунок 2.1 – Схема матричного алгоритма нахождения расстояния Левенштейна

2.4 Разработка алгоритмов нахождения расстояния Дамерау-Левенштейна

На рисунке 2.2 приведена схема алгоритма нахождения расстояния Дамерау-Левенштейна с заполнением матрицы. На рисунке 2.3 приведена схема алгоритма нахождения расстояния Дамерау-Левенштейна рекурсивно.

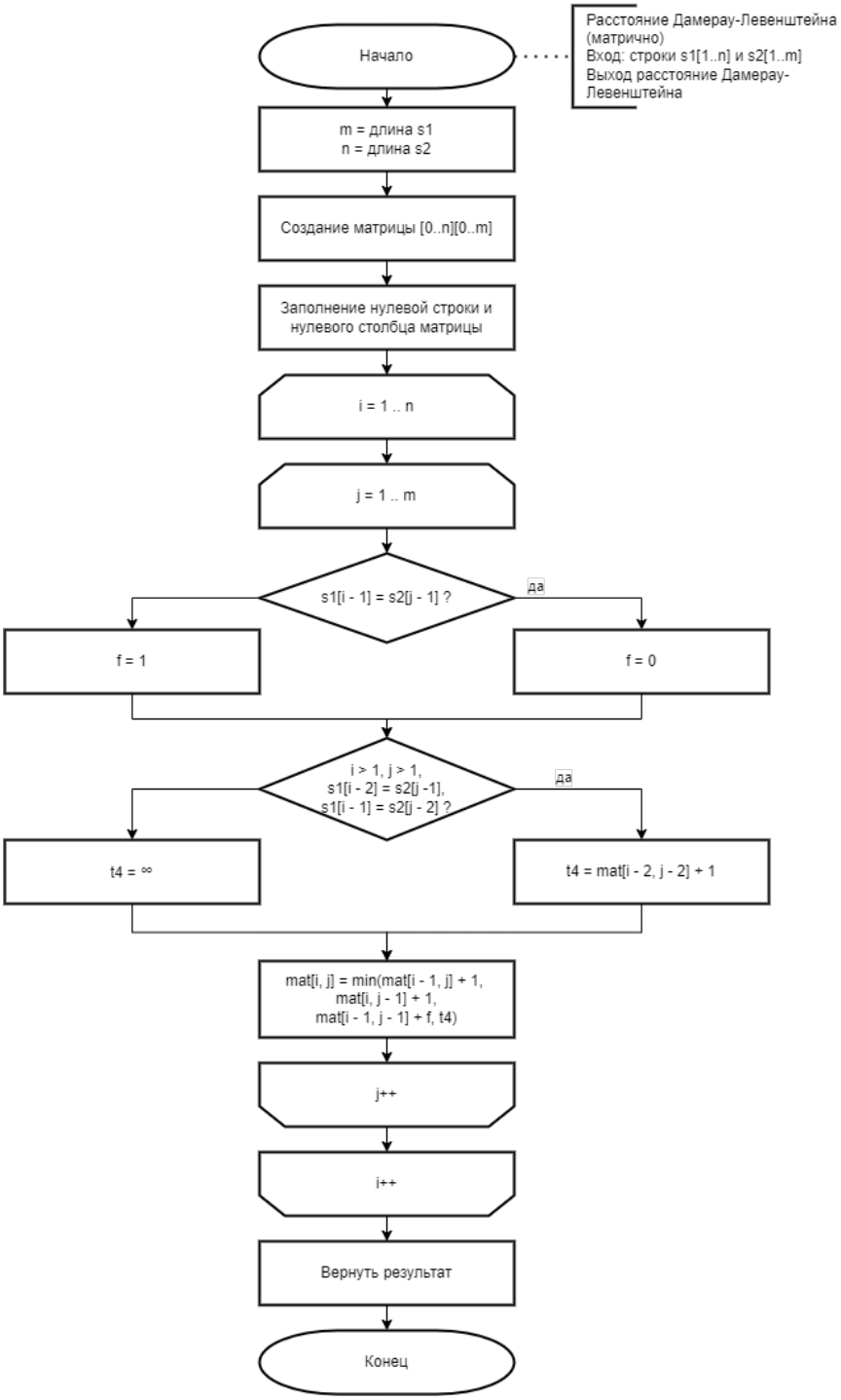

Рисунок 2.2 – Схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна

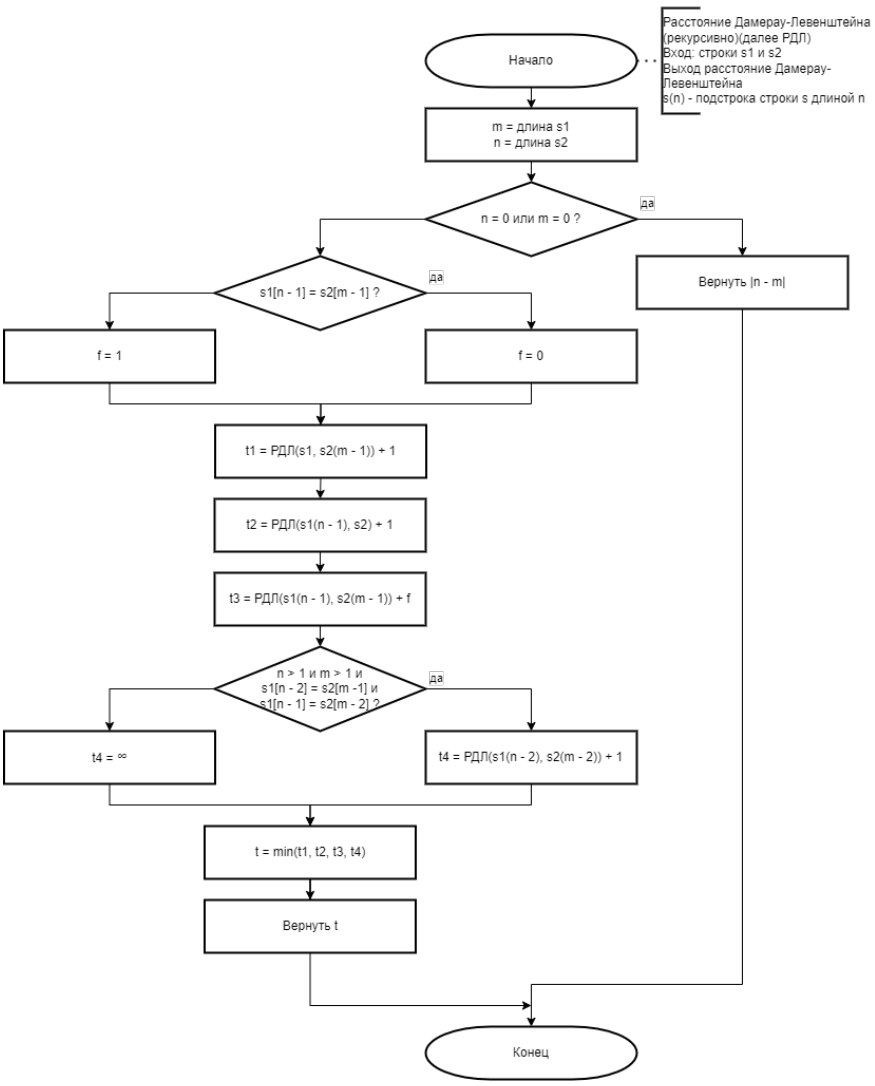

Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

Вывод

Перечислены требования к вводу и программе, а также на основе теоретических данных, полученных из аналитического раздела были построены схемы требуемых алгоритмов.

3 Технологическая часть

В данном разделе будут приведены требования к программному обеспечению, средства реализации и листинги кода.

3.1 Требования к ПО

К программе предъявляется ряд требований:

- у пользователя есть выбор алгоритма, или какой-то один, или все сразу, а также есть выбор тестирования времени;
- на вход подаются две строки на русском или английском языке в любом регистре;
- на выходе — искомое расстояние для выбранного метода (выбранных методов) и матрицы расстояний для матричных реализаций.

3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язык программирования C# [3].

Язык C# является полностью объектно-ориентированным. Все необходимые библиотеки для реализации поставленной задачи являются стандартными.

Время работы алгоритмов было замерено с помощью функции clock() [4].

3.3 Сведения о модулях программы

Программа состоит из семи модулей.

1. Programm.cs – главный файл программы, в котором располагается код меню;

2. BaseAlgo.cs, BaseRecAlgo.cs – файлы с базовыми классами алгоритмов.

3. DamLevAlgo.cs, RecDamLevAlgo.cs, LevensteinAlgo.cs, RecCacheDamLevAlgo.cs – файлы с кодами алгоритмов

3.4 Реализации алгоритмов

В листингах 3.1, 3.2, 3.3, 3.4 приведены реализации алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

Листинг 3.1 – Класс с алгоритмом нахождения расстояния Левенштейна.

```
internal class LevensteinAlgo : BaseAlgo
{
    public override int[,] DoAlgorithm(string f_str, string s_str)
    {
        int[,] matrix = new int[s_str.Length + 1, f_str.Length +
            1];
        for (int i = 0; i < f_str.Length + 1; i++)
        matrix[0, i] = i;
        for (int i = 0; i < s_str.Length + 1; i++)
        matrix[i, 0] = i;
        for (int i = 1; i < s_str.Length + 1; i++)
        for (int j = 1; j < f_str.Length + 1; j++)
        {
            int t1 = matrix[i, j - 1] + 1;
            int t2 = matrix[i - 1, j] + 1;
            int t3 = matrix[i - 1, j - 1] + (s_str[i - 1] ==
                f_str[j - 1] ? 0 : 1);
            matrix[i, j] = Math.Min(t1, Math.Min(t2, t3));
        }
        return matrix;
    }
}
```

Листинг 3.2 – Класс с алгоритмом нахождения расстояния Дамерау-Левенштейна.

```
internal class DamLevAlgo : BaseAlgo
{
```

```
3       public override int[,] DoAlgorithm(string f_str, string s_str)
4       {
5           int[,] matrix = new int[s_str.Length + 1, f_str.Length +
                1];
6           for (int i = 0; i < f_str.Length + 1; i++)
7           matrix[0, i] = i;
8           for (int i = 0; i < s_str.Length + 1; i++)
9           matrix[i, 0] = i;
10          for (int i = 1; i < s_str.Length + 1; i++)
11          for (int j = 1; j < f_str.Length + 1; j++)
12          {
13              int t1 = matrix[i, j - 1] + 1;
14              int t2 = matrix[i - 1, j] + 1;
15              int t3 = matrix[i - 1, j - 1] + (s_str[i - 1] ==
                    f_str[j - 1] ? 0 : 1);
16              int t4 = int.MaxValue;
17              if (i > 1 && j > 1 && s_str[i - 1] == f_str[j - 2] &&
                    s_str[i - 2] == f_str[j - 1])
18              t4 = matrix[i - 2, j - 2] + 1;
19              matrix[i, j] = Math.Min(Math.Min(t1, t4),
                    Math.Min(t2, t3));
20          }
21          return matrix;
22      }
23  }
```

Листинг 3.3 – Класс с алгоритмом нахождения расстояния Дамерау-Левенштейна с использованием рекурсии.

```
1   internal class RecDamLevAlgo : BaseRecurAlgo
2   {
3       public override int DoAlgorithm(string f_str, string s_str)
4       {
5           int n = f_str.Length, m = s_str.Length;
6           if (n == 0 || m == 0)
7           return Math.Abs(n - m);
8           int t1 = DoAlgorithm(f_str[..(n - 1)], s_str[..m]) + 1;
9           int t2 = DoAlgorithm(f_str[..n], s_str[..(m - 1)]) + 1;
10          int t3 = DoAlgorithm(f_str[..(n - 1)], s_str[..(m - 1)])
                + (s_str[^1] == f_str[^1] ? 0 : 1);
11          int t4 = int.MaxValue;
12          if (m > 1 && n > 1 && s_str[^1] == f_str[^2] && s_str[^2]
```

```
            == f_str[^1])
13          t4 = DoAlgorithm(f_str[..(n - 2)], s_str[..(m - 2)]) + 1;
14          return Math.Min(Math.Min(t1, t4), Math.Min(t2, t3));
15      }
16  }
```

Листинг 3.4 – Класс с алгоритмом нахождения расстояния Левенштейна с использованием рекурсии.

```
1   internal class RecCacheDamLevAlgo : BaseRecurAlgo
2   {
3       public override int DoAlgorithm(string f_str, string s_str)
4       {
5           int n = f_str.Length, m = s_str.Length;
6           int[,] matrix = new int[f_str.Length + 1, s_str.Length +
                1];
7
8           static int recursive(string f_str, string s_str, int n,
                int m, int[,] matrix)
9           {
10              if (matrix[n, m] != -1)
11              return matrix[n, m];
12
13              if (n == 0)
14              {
15                  matrix[n, m] = m;
16                  return matrix[n, m];
17              }
18
19              if (n > 0 && m == 0)
20              {
21                  matrix[n, m] = n;
22                  return matrix[n, m];
23              }
24              int delete = recursive(f_str, s_str, n - 1, m,
                    matrix) + 1;
25              int add = recursive(f_str, s_str, n, m - 1, matrix) +
                    1;
26              int change = recursive(f_str, s_str, n - 1, m - 1,
                    matrix) + (s_str[m - 1] == f_str[n - 1] ? 0 : 1);
27              int xch = int.MaxValue;
28              if (m > 1 && n > 1 && s_str[m - 1] == f_str[n - 2] &&
```

```
                s_str[m - 2] == f_str[n - 1])
29              xch = recursive(f_str, s_str, n - 2, m - 2, matrix) +
                    1;
30
31              matrix[n, m] = Math.Min(Math.Min(add, xch),
                    Math.Min(delete, change));
32
33              return matrix[n, m];
34          }
35
36          for (int i = 0; i < n + 1; i++)
37          for (int j = 0; j < m + 1; j++)
38          matrix[i, j] = -1;
39
40          recursive(f_str, s_str, n, m, matrix);
41
42          return matrix[n, m];
43
44      }
45  }
```

3.5 Функциональные тесты

В таблице 3.1 приведены функциональные тесты для алгоритмов вычисления расстояния Левенштейна (в таблице столбец подписан "Левенштейн") и Дамерау — Левенштейна (в таблице - "Дамерау-Л."). Тестирование проводилось по методу черного ящика. Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты

Входные данные			Ожидаемый результат	
№	Строка 1	Строка 2	Левенштейн	Дамерау-Л.
1	скат	кот	2	2
2	машина	малина	1	1
3	дворик	доврик	2	1
4	λ	университет	11	11
5	сентябрь	λ	8	8
8	тело	телодвижение	8	8
9	ноутбук	планшет	7	7
10	глина	малина	2	2
11	рекурсия	ркерусия	3	2
12	браузер	баурзер	2	2
13	bring	brought	4	4
14	moment	minute	4	4
15	person	eye	5	5
16	week	weekend	3	3
17	city	town	4	4

Вывод

Были разработаны и протестированы реализации алгоритмов: нахождения расстояния Дамерау-Левенштейна рекурсивно, с заполнением матрицы и рекурсивно с заполнением матрицы, а также нахождения расстояния Левенштейна с матрицей.

4 Исследовательская часть

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

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось исследование:

- операционная система Windows 10 Корпоративная, Версия 21H1, Сборка ОС 19043.2006;
- память 8 ГБ;
- процессор AMD Ryzen 5 4600H с видеокартой Radeon Graphics 3.00 ГГц [5].

Исследование проводилось на ноутбуке, включенном в сеть электропитания. Во время исследования ноутбук был нагружен только встроенными приложениями окружения, а также непосредственно системой.

4.2 Демонстрация работы программы

На рис. 4.1 приведен пример работы программы: пользователь выбрал первый пункт меню – расчёт расстояния Левенштейна – и ввёл две строки. Отображено результирующее расстояние, а также матрица промежуточных расстояний между подстроками, поскольку выбранный алгоритм расчёта основан на заполнении матрицы по рекуррентной формуле 1.1.

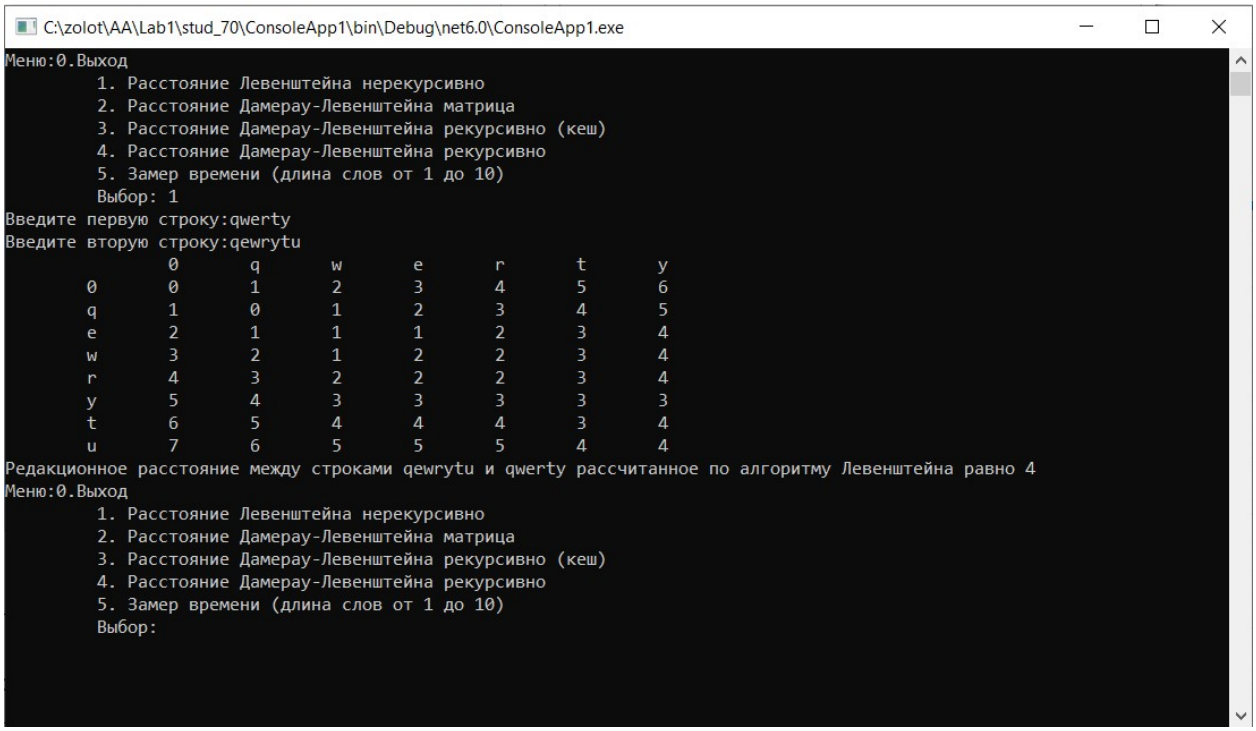

Рисунок 4.1 – Демонстрация работы программы

4.3 Время выполнения реализаций алгоритмов

Замеры процессорного времени выполнения реализаций алгоритмов выполнены при помощи функции clock().[4] Данная функция всегда возвращает значения времени, а именно сумму системного и пользовательского процессорного времени текущего процессора, типа float в тиках.

Замеры времени для каждой длины слов проводились 100 раз. Длины строк совпадают. В качестве результата взято среднее время работы алгоритма на данной длине слова.

Результаты замеров приведены в таблицах 4.1–4.2 (время в тиках) и на рис. 4.2.

Таблица 4.1 – Результаты замеров времени

Длина строки	ЛевМатр	ДамЛевМатр	РекДамЛевМатр
0	0	0	0,001
10	0,003	0,003	0,006
20	0,009	0,01	0,017
30	0,018	0,022	0,041
40	0,031	0,04	0,07
50	0,049	0,064	0,108
60	0,068	0,087	0,154
70	0,092	0,122	0,209
80	0,126	0,155	0,27
90	0,156	0,202	0,342
100	0,192	0,246	0,422

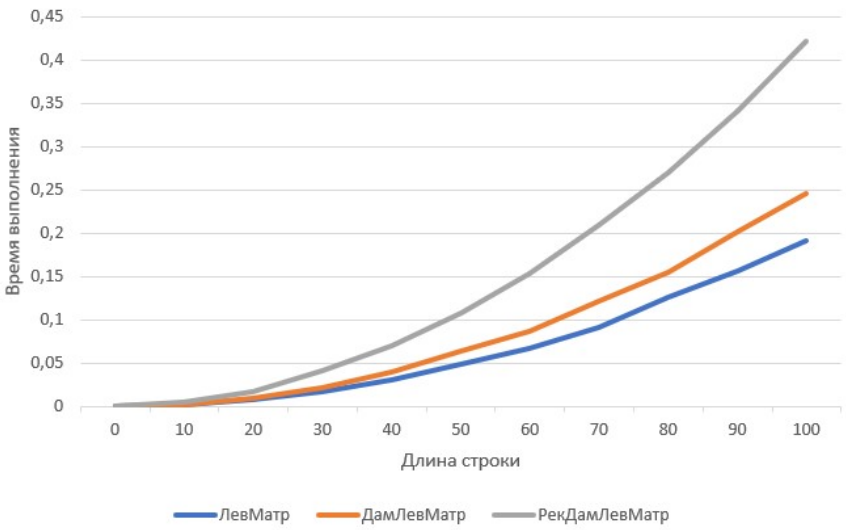

Рисунок 4.2 – Зависимость времени выполнения реализаций алгоритмов от длины строки

Таблица 4.2 – Результаты замеров времени выполнения реализации рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна

Длина строки	РекДамЛев
0	0,01
1	0
2	0
3	0
4	0,02
5	0,08
6	0,39
7	2,02
8	10,6
9	57,48

4.4 Использование памяти

Нерекурсивные алгоритмы нахождения расстояний Дамерау-Левенштейна и Левенштейна не отличаются друг от друга с точки зрения использования памяти, поэтому достаточно рассмотреть лишь разницу рекурсивной и матричной реализаций алгоритмов поиска расстояния Дамерау-Левенштейна.

В C# строки и матрицы являются ссылочными типами данных, поэтому память под них выделяется один раз в куче [6].

Пусть длина строки S1 – n, длина строки S2 – m, тогда затраты памяти на приведенные выше алгоритмы будут следующими.

Матричный алгоритм поиска расстояния Левенштейна:

- ссылки на строки – 16 байт;
- матрица – ((m + 1) * (n + 1)) * sizeof(int);
- вспомогательные переменные – 5 * sizeof(int);

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящий строк.

Рекурсивный алгоритм поиска расстояния (для каждого вызова):

- ссылки на строки – 16 байт;

- вспомогательные переменные – 6 * sizeof(int);
- адрес возврата.

Рекурсивный алгоритм поиска расстояния с использованием кеша (для каждого вызова) V_{call}:

- длины строк – 2 * sizeof(int);
- вспомогательные переменные – 4 * sizeof(int);
- ссылки на строки – 16 байт;
- ссылка на матрицу – 8 байт;
- адрес возврата.

Помимо самих вызовов еще требуется память для хранения самой матрицы: $(m * n) * \text{sizeof(int)}$.

Итоговая память, затрачиваемая реализацией рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна с использованием кеша:
$(m * n) * \text{sizeof(int)} + (m + n) * V_{call}$.

Вывод

Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна работает намного дольше итеративных реализаций. На словах длиной 10 символов, матричная реализация алгоритма нахождения расстояния Левенштейна превосходит по времени работы рекурсивную на несколько порядков.

Рекурсивный алгоритм с заполнением матрицы превосходит простой рекурсивный и работает в 1.5–2 раза дольше по сравнению с матричными алгоритмами. Алгоритм нахождения расстояния Дамерау-Левенштейна по времени выполнения сопоставим с алгоритмом нахождения расстояния Левенштейна. В нём добавлена дополнительная проверка, позволяющая находить ошибки пользователя, связанные с неверным порядком букв, в связи с чем он работает незначительно дольше, чем алгоритм нахождения расстояния Левенштейна.

Но по расходу памяти итеративные алгоритмы проигрывают рекурсивному: максимальный размер используемой памяти в них растёт как произведение длин строк, в то время как у рекурсивного алгоритма — как сумма длин строк.

Заключение

Поставленная цель была достигнута.

В ходе выполнения лабораторной работы были решены следующие задачи:

- изучены алгоритмы нахождения расстояний Левенштейна и Дамерау-Левенштейна;
- для некоторых реализаций применены методы динамического программирования, что позволило сделать алгоритмы быстрее;
- реализованы алгоритмы поиска расстояния Левенштейна с заполнением матрицы, с использованием рекурсии и с помощью рекурсивного заполнения матрицы (рекурсивный с использованием кеша);
- реализованы алгоритмы поиска расстояния Дамерау-Левенштейна с использованием рекурсии;
- проведен сравнительный анализ линейной и рекурсивной реализаций алгоритмов определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);
- проведено экспериментальное подтвержение различий во временной эффективности рекурсивной и нерекурсивной реализаций алгоритмов при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализации на различных длинах строк;
- подготовлен отчет о лабораторной работе.

Экспериментально было подтверждено различие во временной эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определенного между строками при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализаций на различных длин строк.

В результате исследований можно прийти к выводу, что матричная реализация алгоритмов нахождения расстояний заметно выигрывает по времени при росте длины строк, но проигрывает по количеству затрачиваемой памяти.

Список использованных источников

[1] Левенштейн В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов / Доклады АН СССР, т. 163 – М. : Наука, 1965. С. 845–848.

[2] Черненький В. М. Гапанюк Ю. Е. Методика идентификации пассажира по установочным данным. М.: Вестник МГТУ им. Н.Э. Баумана. Сер. “Приборостроение”, 2012. Т. 163. С. 30–34.

[3] Краткий обзор языка C# [Электронный ресурс]. Режим доступа: `https://learn.microsoft.com/ru-ru/dotnet/csharp/tour-of-csharp/` (дата обращения: 25.09.2022).

[4] ISO/IEC 9899:1999 [Электронный ресурс]. Режим доступа: `https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf` Глава 7.23.2.1 The clock function (дата обращения: 25.09.2022).

[5] AMD Ryzen™ 5 4600H [Электронный ресурс]. Режим доступа: `https://www.amd.com/ru/products/apu/amd-ryzen-5-4600h` (дата обращения: 25.09.2022).

[6] Ссылочные типы. Режим доступа: `https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/reference-types` (дата обращения: 25.09.2022).