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»

на тему: «Расстояние Левенштейна и Дамерау-Левенштейна»

Студент ИУ7-53Б _____ Лысцев Н. Д.
(Группа) (Подпись, дата) (И. О. Фамилия)

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

2023 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ **4**

1 Аналитический раздел **6**

1.1 Алгоритм поиска расстояния Левенштейна 6

1.1.1 Нерекурсивый алгоритм поиска расстояния Левенштейна . 7

1.2 Алгоритм поиска расстояния Дамерау-Левенштейна 8

1.2.1 Нерекурсивый алгоритм поиска расстояния Дамерау-Левенштейна 8

1.2.2 Рекурсивый алгоритм поиска расстояния Дамерау-Левенштейна 9

1.2.3 Рекурсивый алгоритм поиска расстояния Дамерау-Левенштейна с кешированием 9

2 Конструкторский раздел **10**

2.1 Разработка алгоритма поиска расстояния Левенштейна 10

2.2 Разработка алгоритма поиска расстояния Дамерау- Левенштейна 12

3 Технологический раздел **17**

3.1 Требования к программному обепечению 17

3.1.1 Требования к вводу 17

3.1.2 Требования к программе 17

3.2 Средства реализации 18

3.3 Сведения о модулях программы 18

3.4 Реализации алгоритмов 19

3.5 Функциональные тесты 23

4 Исследовательский раздел **24**

4.1 Технические характеристики 24

4.2 Время выполнения алгоритмов 24

4.3 Использование памяти 26

ЗАКЛЮЧЕНИЕ **32**

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ **33**

ВВЕДЕНИЕ

Расстояние Левенштейна (редакционное расстояние, дистанция редактирования) — метрика, определяющаяся как минимальное количество односимвольных операций (а именно вставки, удаления, замены), необходимых для превращения одной последовательности символов в другую.

Расстояние Левенштейна и Дамерарау-Левенштейна активно применяется для исправления ошибок в слове (в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи), для сравнения текстовых файлов, в биоинформатике для сравнения генов, хромосом и белков.

Расстояние Дамерау-Левенштейна — модификация расстояния Левенштейна. К операциям вставки, удаления и замены символов добавляется операция транспозиции (перестановки) двух соседних символов.

Целью данной лабораторной работы является изучение, описание, реализация и исследование алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.

Для достижения поставленной цели необходимо решить следующие задачи:

1) Изучить и 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 алгоритм поиска расстояния Дамерау-Левенштейна с кешированием.

3) Провести анализ эффективности реализаций алгоритмов по памяти и по времени.

4) Обосновать полученные результаты в отчете к выполненной лабораторной работе.

1 Аналитический раздел

В этом разделе будет дано описание алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.

1.1 Алгоритм поиска расстояния Левенштейна

Расстояние Левенштейна [1], редакционное расстояние — метрика сходства между двумя строковыми последовательностями, минимальное количество редакторский операций вставки, удаления, замены символа, необходимых для превращения одной строки в другую.

Каждая редакторская операция имеет свою цену. Для алгоритма поиска расстояния Левенштейна устанавливаются следующие цены:

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

   $w(a, a) = 0$ — замены не происходит.

2) $w(\varepsilon, b) = 1$ — цена вставки символа b;

3) $w(a, \varepsilon) = 1$ — цена удаления символа a.

Пусть S_1 – первая строка, тогда ее длина будет равна L_1 и пусть S_2 – вторая строка, имеющая длину L_2. $S_1[1...i]$ – подстрока S_1 длиной i символов, начиная с первого, $S_2[1...j]$ – подстрока S_2 длиной j символов.

Введем функцию $D(i, j)$, результатом работы которой является редакционное расстояние между двумя подстроками $S_1[1...i]$ $S_2[1...j]$.

Функция D определяется следующей рекуррентной формулой:

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

где сравнение символов строк S_1 и S_2 рассчитывается как:

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

Тогда расстояние Левенштейна $d(S_1, S_2)$ между двумя строками S_1 и S_2 будет равно $D(L_1, L_2)$.

1.1.1 Нерекурсивый алгоритм поиска расстояния Левенштейна

Формула 1.1 является рекуррентной. С ростом i, j растет время выполнения программы, так как во время работы множества промежуточных значений $D(i, j)$ вычисляются по нескольку раз.

Для оптимизации времени работы алгоритма можно использовать матрицу для хранения промежуточных значений. Эта матрица имеет размеры $(L_1 + 1) \times (L_2 + 1)$. Значения в ячейке $[i, j]$ равно значению $D(i, j)$. Вся матрица заполняется в соответствии с формулой 1.1.

Алгоритм, использующий матрицу, будет неэффективен по памяти при больших L_1 и L_2, поскольку вычисленные промежуточные значения будут храниться в памяти после их использования.

Заметим, что для задачи поиска расстояния Левенштейна в этом алгоритме нет необходимости хранить всю матрицу целиком — достаточно лишь

текущей и предыдущей строк. Это существенно уменьшит потребляемую память с сохранением хорошей скорости работы.

1.2 Алгоритм поиска расстояния Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна [2] — модификация расстояния Левенштейна. К операциям вставки, удаления и замены символа добавляется операция транспозиции (перестановки) двух соседних символов.

Расстояние Дамерау-Левенштейна может быть найдено по формуле 1.3.

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

1.2.1 Нерекурсивый алгоритм поиска расстояния Дамерау-Левенштейна

Формула 1.3 является рекуррентной. С ростом i, j растет время выполнения программы, так как во время работы множества промежуточных значений $d_{a,b}(i,j)$ вычисляются по нескольку раз.

Для оптимизации времени работы алгоритма можно использовать матрицу для хранения промежуточных значений. Эта матрица имеет размеры $(L_1+1) \times (L_2+1)$, где L_1 и L_2 — длины первой и второй строк соответственно. Значения в ячейке $[i,j]$ равно значению $d_{a,b}(i,j)$. Вся матрица заполняется в соответствии с формулой 1.3.

Алгоритм, использующий матрицу, будет неэффективен по памяти при больших L_1 и L_2, поскольку вычисленные промежуточные значения будут храниться в памяти после их использования.

1.2.2 Рекурсивый алгоритм поиска расстояния Дамерау-Левенштейна

Как и в случае с формулой 1.1 формула 1.3 является рекуррентной. Прямая реализация данного алгоритма будет неэффективной по времени из-за того, что промежуточные значения $d_{a,b}(i, j)$ вычисляются не по одному разу в ходе работы алгоритма.

1.2.3 Рекурсивый алгоритм поиска расстояния Дамерау-Левенштейна с кешированием

Идея рекурсивного алгоритма Дамерау-Левенштейна с кешированием заключается в том, чтобы эффективно вычислить минимальное расстояние между двумя строками, используя рекурсивные вызовы. При этом сохраняются результаты вычислений для каждой пары индексов в кеше, чтобы избежать повторных вычислений. Кешем является двумерная матрица размерами $(L_1 + 1) \times (L_2 + 1)$, где L_1 и L_2 — длины первой и второй строк соответственно.

Это позволяет существенно улучшить производительность алгоритма, особенно в случае, когда вычисления могут быть пересчитаны многократно для одних и тех же подстрок.

Вывод

В данном разделе были рассмотрены понятия расстояний Левенштейна и Дамерау-Левенштейна, объяснена разница между этими понятиями, были приведены формулы, описывающие алгоритмы поиска этих расстояний, а также были рассмотрены различные способы их реализации.

2 Конструкторский раздел

В данном разделе будут разработаны алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна и приведены схемы алгоритмов различных способов их реализации.

2.1 Разработка алгоритма поиска расстояния Левенштейна

На рисунках 2.1 и 2.2 приведена схема нерекурсивной реализации алгоритма нахождения расстояния Левенштейна.

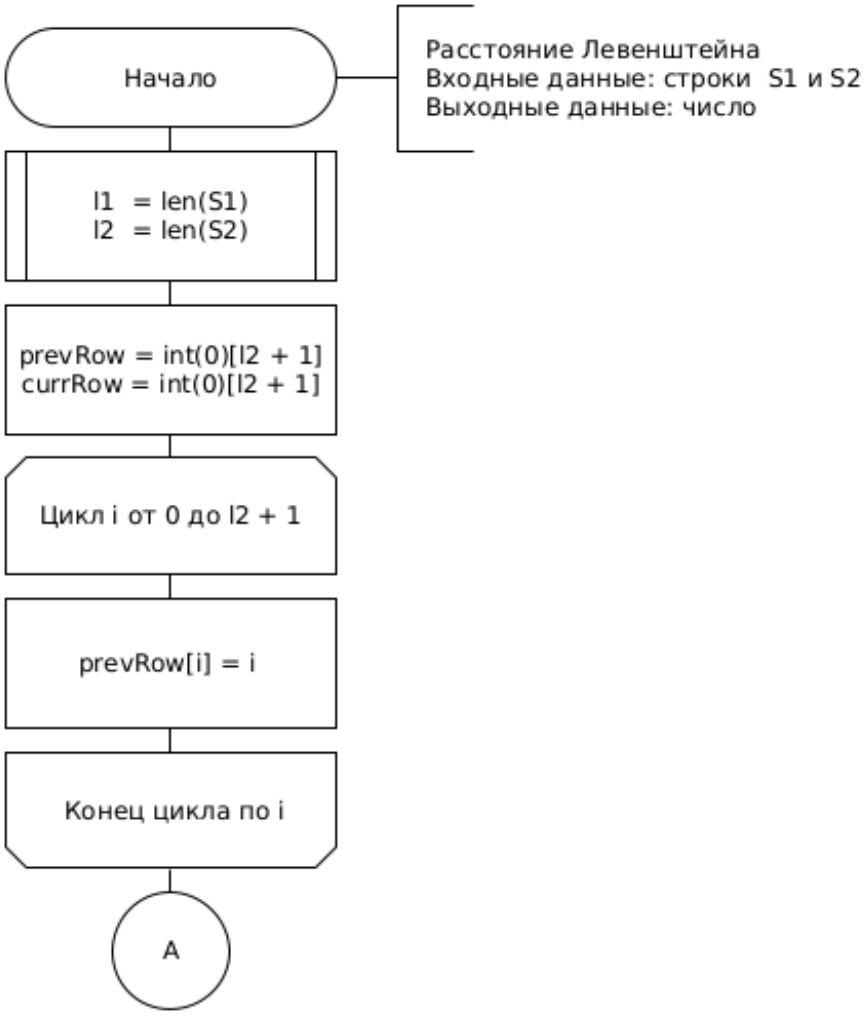

Рисунок 2.1 – Схема нерекурсивного алгоритма нахождения расстояния Левенштейна, первая часть

Рисунок 2.2 – Схема нерекурсивного алгоритма нахождения расстояния Левенштейна, вторая часть

2.2 Разработка алгоритма поиска расстояния Дамерау-Левенштейна

На рисунках 2.3 и 2.4 приведена схема нерекурсивной реализации алгоритма нахождения расстояния Дамерау-Левенштейна.

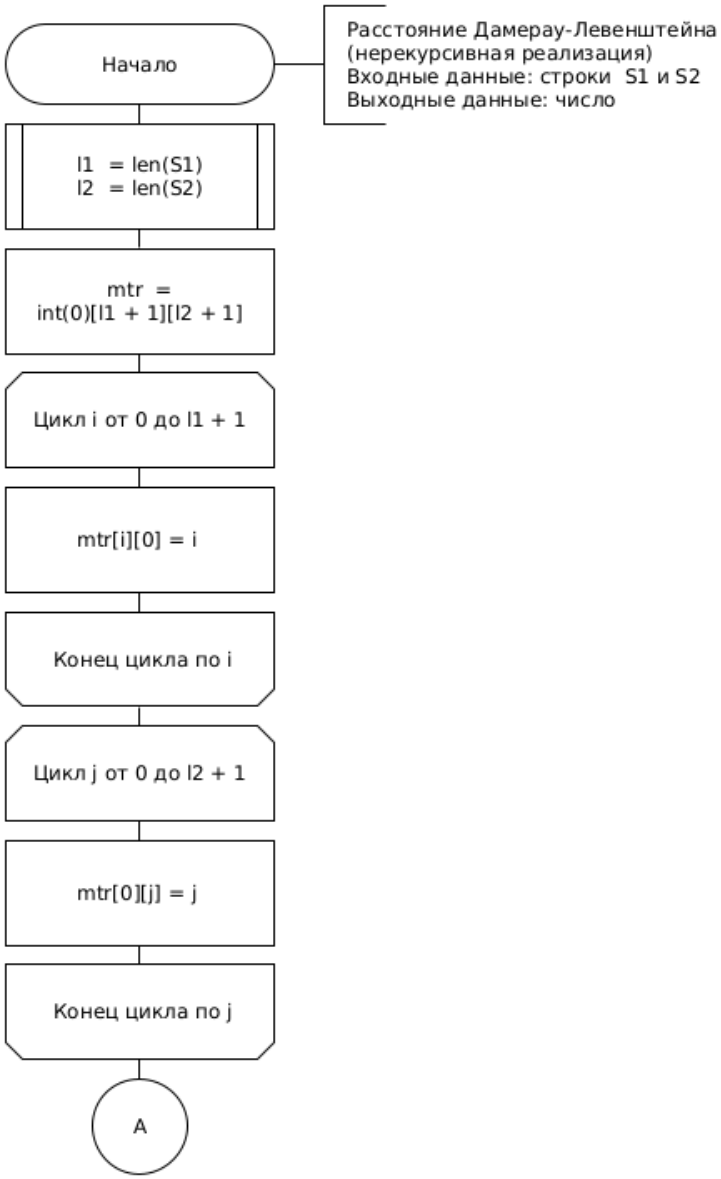

Рисунок 2.3 – Схема нерекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна, первая часть

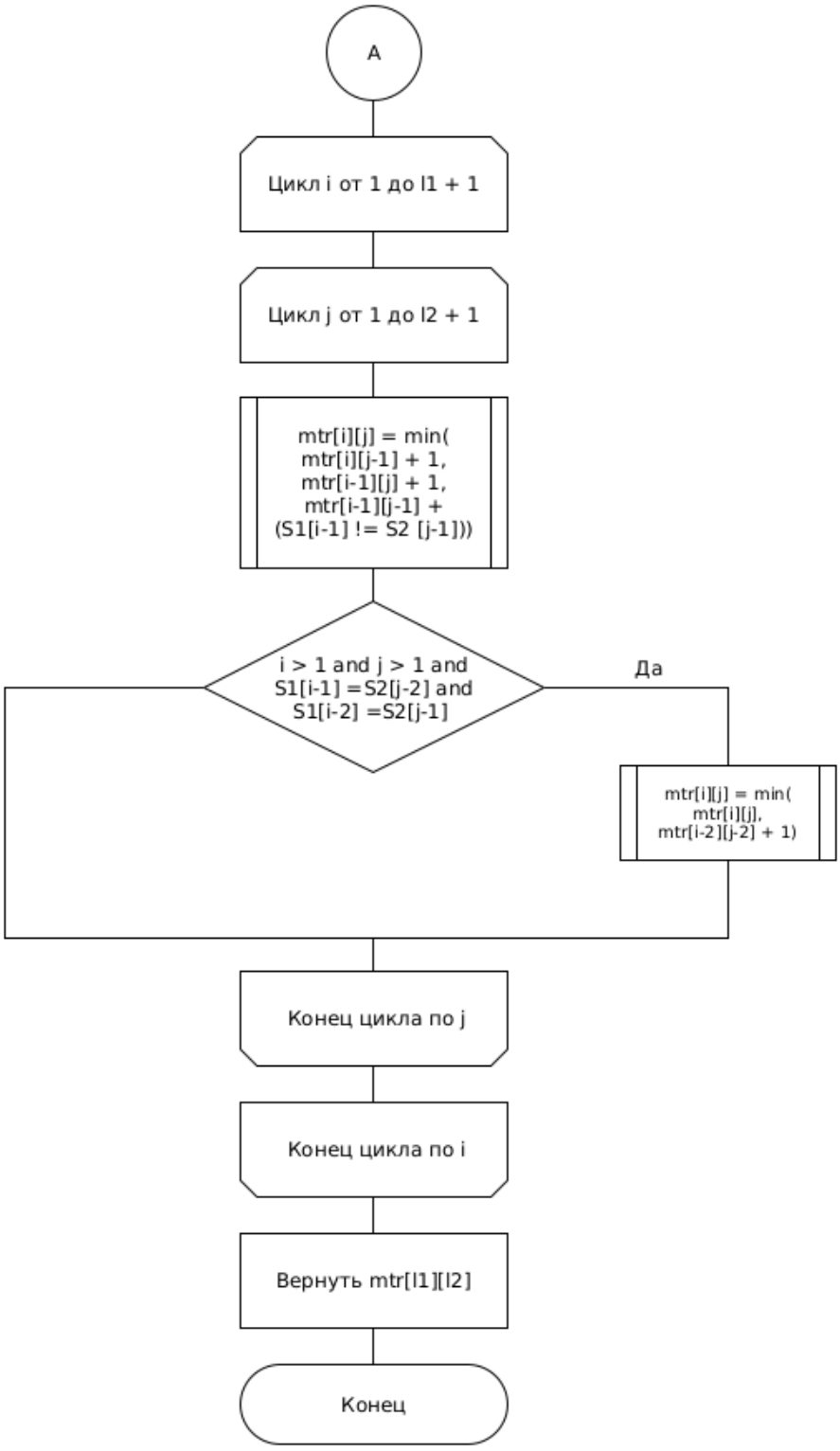

Рисунок 2.4 – Схема нерекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна, вторая часть

На рисунке 2.5 приведена схема рекурсивной реализации алгоритма нахождения расстояния Дамерау-Левенштейна.

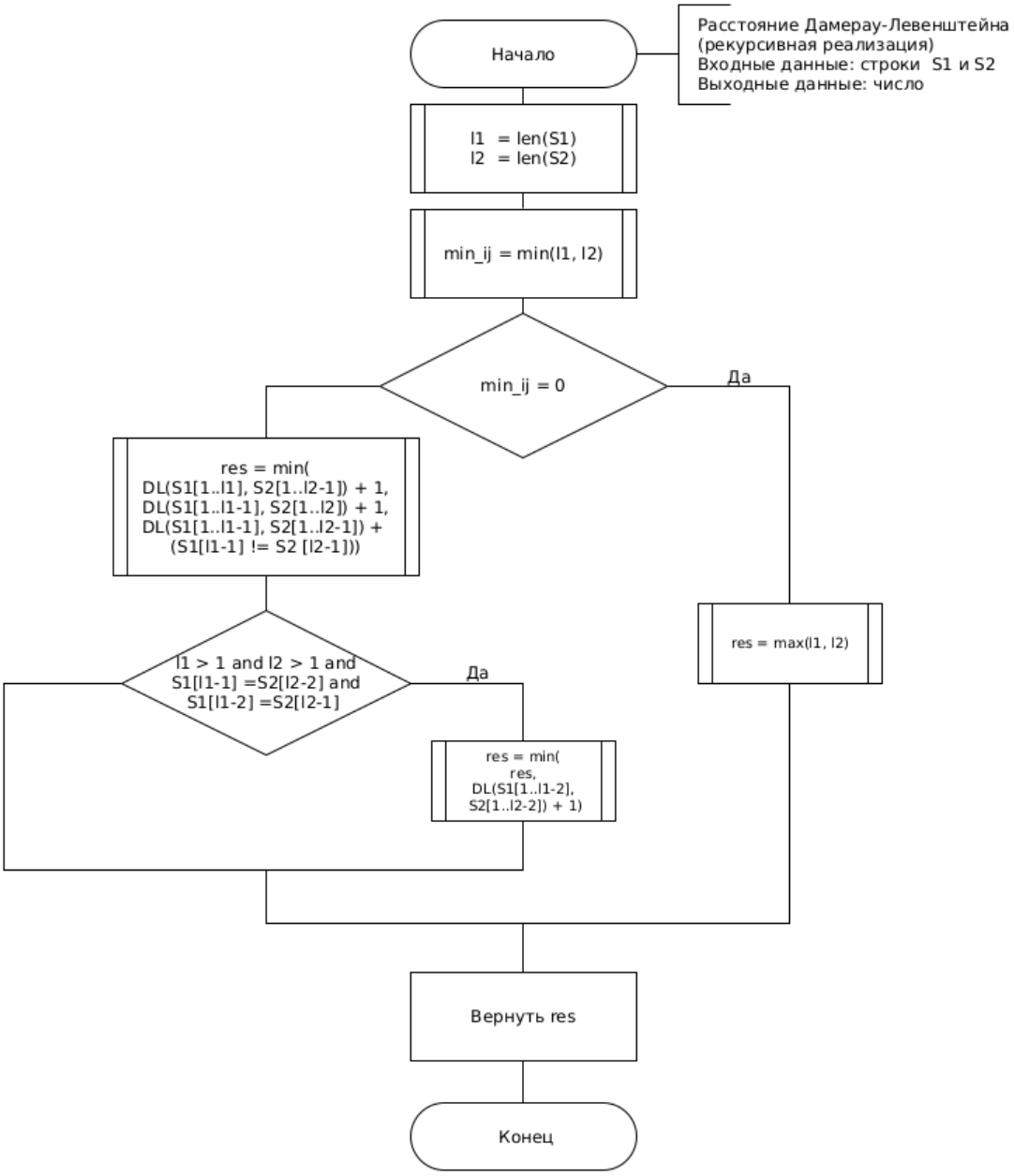

Рисунок 2.5 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

На рисунке 2.6 приведена схема рекурсивной реализации алгоритма нахождения расстояния Дамерау-Левенштейна с кешированием.

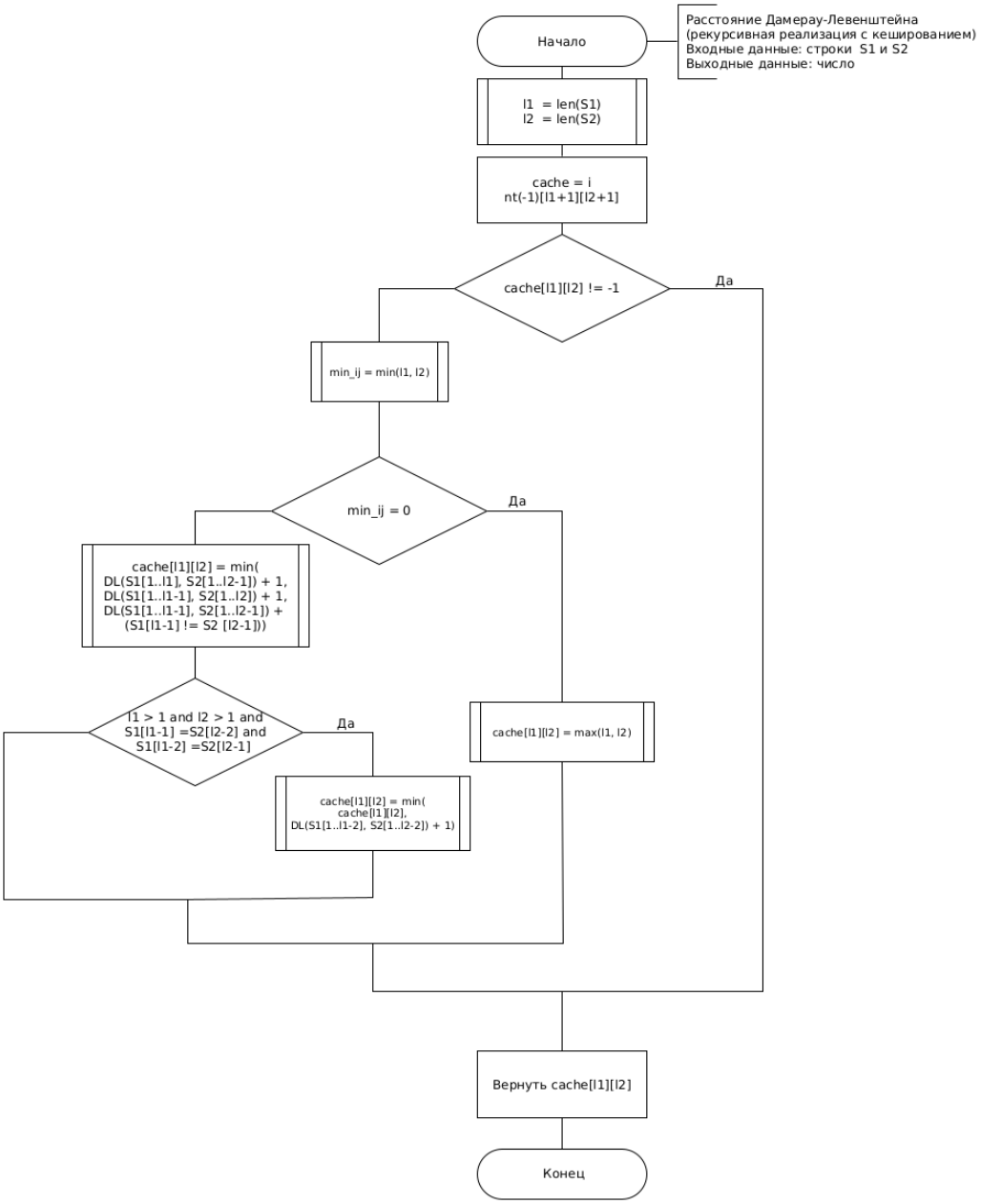

Рисунок 2.6 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с кешированием

Вывод

В данном разделе были перечислены основные требования к программному обеспечению, а также, на основе данных, полученных в аналитическом разделе, были построены схемы алгоритмов различных способов реализации алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.

3 Технологический раздел

В данном разделе будут перечислены требования к реализуемому программному обеспечению, средства реализации, листинги кода и функциональные тесты.

3.1 Требования к программному обепечению

3.1.1 Требования к вводу

К входным данным предъявляется несколько условий:

– на вход подаются две строки;

– входные строки могут содержать цифры, буквы как русского так и английского алфавита;

– верхний и нижний регистр букв считается различным.

3.1.2 Требования к программе

Программа должна удовлетворять следующим условиям:

– пустые последовательности строк должны корректно обрабатываться;

– результат работы программы – число (расстояние Левенштейна или Дамерау-Левенштейна);

– наличие консольного интерфейса взаимодействия с программой;

– наличие функционала для замеров процессорного времени и оценки затрат по памяти реализаций алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.

3.2 Средства реализации

В качестве языка программирования для этой лабораторной работы был выбран $C++$ [3] по следующим причинам:

- в $C++$ есть встроенный модуль *ctime*, предоставляющий необходимый функционал для замеров процессорного времени;
- в стандартной библиотеке $C++$ есть оператор $sizeof$, позволяющий получить размер переданного объекта в байтах. Следовательно, $C++$ предоставляет возможности для проведения точных оценок по используемой памяти;
- в $C++$ есть тип данных $std :: wstring$, который позволяет хранить и использовать как кириллические, так и латинские символы.

В качестве функции, которая будет осуществлять замеры процессорного времени, будет использована функция $clock\_gettime$ из встроенного модуля *ctime* [4].

3.3 Сведения о модулях программы

Программа состоит из семи модулей:

1) `algorithms.cpp` — модуль, хранящий реализации алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна;
2) `processTime.cpp` — модуль, содержащий функцию для замера процессорного времени;
3) `memoryMeasurements.cpp` — модуль, содержащий функции, позволяющие провести сравнительный анализ использования памяти в реализациях алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна;
4) `timeMeasurements.cpp` — модуль, содержащий функции, позволяющие провести сравнительный анализ использования времени в реализациях алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна;

5) `main.cpp` — файл, содержащий точку входа в программу, из которой происходит вызов алгоритмов по разработанному интерфейсу;

6) `interface.cpp` — модуль, содержащий функции для обработки действий пользователя при взаимодействии с программой;

7) `task7` — модуль, содержащий набор скриптов для проведения замеров программы по времени и памяти и построения графиков по полученным данным.

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 – Функция нахоxжения расстояния Левенштейна нерекурсивным способом

```
int distanceNotRecur(const std::wstring &str1, const std::wstring
   &str2)
{
    size_t l1 = str1.length(), l2 = str2.length();
    int currRow[l2 + 1] = {}, prevRow[l2 + 1] = {};
    for (size_t i = 0; i <= l2; ++i)
        prevRow[i] = i;
    for (size_t i = 1; i <= l1; ++i)
    {
        currRow[0] = i;
        for (size_t j = 1; j <= l2; ++j)
        {
            int isEqual = str1[i - 1] == str2[j - 1] ? 0 : 1;
            currRow[j] = std::min({currRow[j - 1] + 1, prevRow[j]
               + 1, prevRow[j - 1] + isEqual});
        }
        swapArrays(prevRow, currRow, l2 + 1);
    }
    return prevRow[l2];
}
```

Листинг 3.2 – Функция нахожжения расстояния Дамерау-Левенштейна нерекурекурсивным способом

```
int distanceNotRecurWithMatrix(const std::wstring &str1, const
   std::wstring &str2)
{
    size_t l1 = str1.length(), l2 = str2.length();
    int **mtr = matrix_alloc(l1 + 1, l2 + 1);
    for (size_t i = 0; i <= l1; ++i)
        mtr[i][0] = i;
    for (size_t j = 0; j <= l2; ++j)
        mtr[0][j] = j;
    for (size_t i = 1; i <= l1; ++i)
        for (size_t j = 1; j <= l2; ++j)
        {
            int isEqual = str1[i - 1] == str2[j - 1] ? 0 : 1;
            mtr[i][j] = std::min({mtr[i][j - 1] + 1,
                                  mtr[i - 1][j] + 1,
                                  mtr[i - 1][j - 1] + isEqual});
            if (i > 1 && j > 1 &&
                str1[i - 1] == str2[j - 2] &&
                str1[i - 2] == str2[j - 1])
                mtr[i][j] = std::min(mtr[i][j], mtr[i - 2][j - 2]
                   + 1);
        }
    int res = mtr[l1][l2];
    matrix_free(mtr, l1 + 1);
    return res;
}
```

Листинг 3.3 – Функция нахохжения расстояния Дамерау-Левенштейна рекурсивным способом

```
static int _distanceRecur(const std::wstring &str1, const
   std::wstring &str2, const size_t i, const size_t j)
{
    if (std::min(i, j) == 0)
        return std::max(i, j);
    int isEqual = str1[i - 1] == str2[j - 1] ? 0 : 1;
    int tmp_res = std::min({_distanceRecur(str1, str2, i, j - 1) +
       1,
                            _distanceRecur(str1, str2, i - 1, j) +
                               1,
                            _distanceRecur(str1, str2, i - 1, j -
                               1) + isEqual});
    if (i > 1 && j > 1 &&
        str1[i - 1] == str2[j - 2] &&
        str1[i - 2] == str2[j - 1])
        tmp_res = std::min(tmp_res, _distanceRecur(str1, str2, i -
           2, j - 2) + 1);
    return tmp_res;
}

int distanceRecur(const std::wstring &str1, const std::wstring
   &str2)
{
    size_t l1 = str1.length(), l2 = str2.length();
    return _distanceRecur(str1, str2, l1, l2);
}
```

Листинг 3.4 – Функция нахохжения расстояния Дамерау-Левенштейна рекурсивным способом с кеширование

```
static int _distanceRecurWithCaching(const std::wstring &str1,
   const std::wstring &str2, const size_t i, const size_t j, int
   **cache)
{
    if (cache[i][j] != -1)
        return cache[i][j];
    if (std::min(i, j) == 0)
        return std::max(i, j);
    int isEqual = str1[i - 1] == str2[j - 1] ? 0 : 1;
    cache[i][j] = std::min({_distanceRecurWithCaching(str1, str2,
      i, j - 1, cache) + 1,
                            _distanceRecurWithCaching(str1, str2,
                               i - 1, j, cache) + 1,
                            _distanceRecurWithCaching(str1, str2,
                               i - 1, j - 1, cache) + isEqual});
    if (i > 1 && j > 1 &&
        str1[i - 1] == str2[j - 2] &&
        str1[i - 2] == str2[j - 1])
        cache[i][j] = std::min(cache[i][j],
           _distanceRecurWithCaching(str1, str2, i - 2, j - 2,
           cache) + 1);
    return cache[i][j];
}

int distanceRecurWithCaching(const std::wstring &str1, const
   std::wstring &str2)
{
    size_t l1 = str1.length(), l2 = str2.length();
    int **cache = matrix_alloc(l1 + 1, l2 + 1, -1);
    int res = _distanceRecurWithCaching(str1, str2, l1, l2, cache);
    matrix_free(cache, l1 + 1);
    return res;
}
```

3.5 Функциональные тесты

В таблице 3.1 приведены тестовые данные, на которых было протестировано разработанное ПО. Все тесты были успешно пройдены.

Таблица 3.1 – Функциональные тесты

Входные данные		**Расстояние и алгоритм**			
Строка 1	**Строка 2**	**Левенштейна**	**Дамерау-Левенштейна**		
		Итеративный	**Итеративный**	**Рекурсивный**	
				Без кеша	**С кешом**
"пустая"	"пустая"	0	0	0	0
"пустая"	qwerty	6	6	6	6
doctorwho	"пустая"	2	2	2	2
code	decoder	3	3	3	3
transposition	transpose	5	5	5	5
asas молоко	saas	2	1	1	1
друзья	рдузия	3	2	2	2
гибралтар	лабрадор	5	5	5	5
АВТОР	АФФТАР	3	3	3	3
moloko	молоко	6	6	6	6

Вывод

В данном разделе были реализованы и протестированы четыре алгоритма: вычисления расстояния Левенштейна (нерекурсивно), Дамерау-Левенштейна нерекурсивно, рекурсивно, рекурсивно с кешированием.

4 Исследовательский раздел

В данном разделе будут проведены сравнения реализаций алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна по времени работы и по затрачиваемой памяти.

4.1 Технические характеристики

Технические характеристики устройства, на котором проводились исследования:

- операционная система: Ubuntu 22.04.3 LTS x86_64 [5];
- оперативная память: 16 Гб;
- процессор: 11th Gen Intel® Core™ i7-1185G7 @ 3.00GHz × 8.

4.2 Время выполнения алгоритмов

Время работы алгоритмов измерялось с использованием функции *clock_gettime* из встроенного модуля *ctime*.

Замеры времени для каждой длины слов проводились 1000 раз. На вход подавались случайно сгенерированные строки заданной длины.

Результаты замеров реализаций алгоритмов по времени представлены в таблице 4.1. Их графическое представление показано на рисунке 4.1.

Таблица 4.1 – Результаты замеров времени работы алгоритмов для строк с длиной от 1 до 10

Длина, символ	**Время, нс**			
	Левенштейн	**Дамерау-Левенштейн**		
	Итеративный	**Итеративный**	**Рекурсивный**	
			Без кеша	**С кешом**
1	239.336	312.711	208.358	292.637
2	355.681	388.345	416.912	481.743
3	494.631	515.159	1214.162	657.416
4	748.396	821.153	6253.184	1003.308
5	1120.226	1133.970	30624.280	1444.514
6	1479.574	1490.202	160402.600	2073.044
7	2108.166	2012.790	898694.400	2785.704
8	2624.444	2569.196	5036644.000	3623.980
9	3770.614	3869.994	33096280.000	5633.132
10	5394.930	5533.728	202570400.000	10068.464

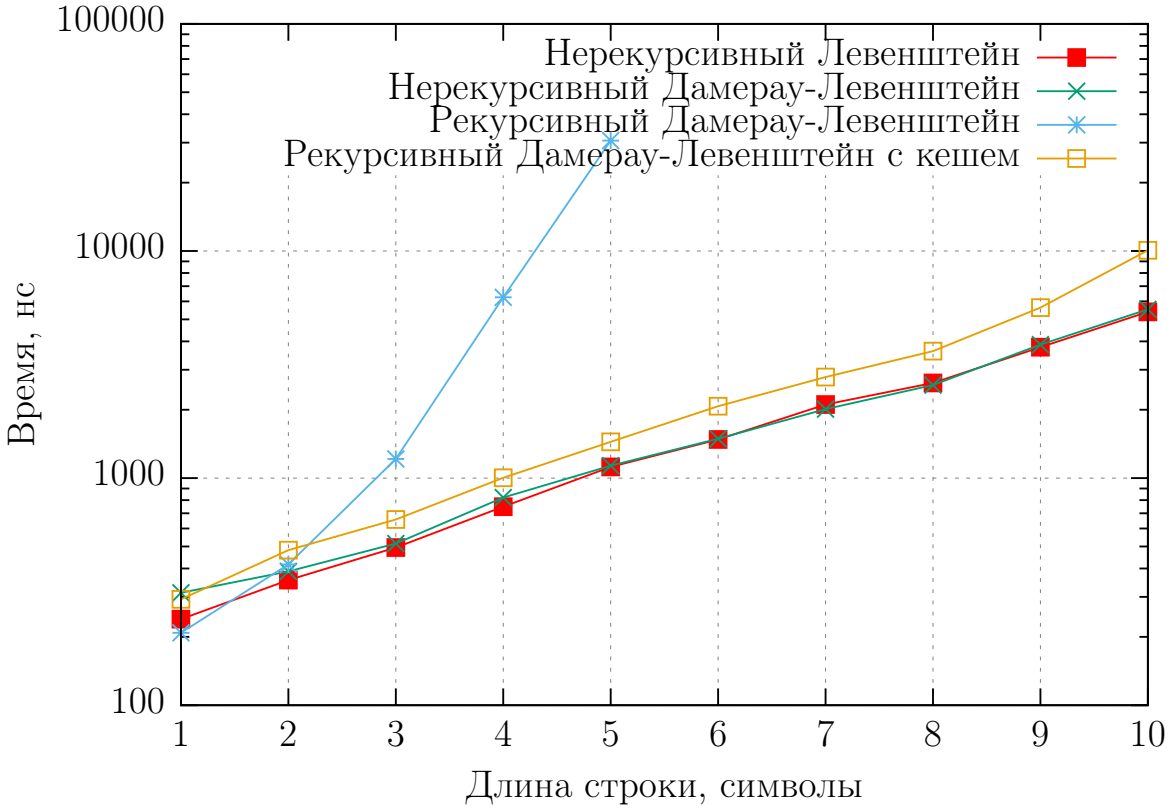

Рисунок 4.1 – Результаты замеров времени работы алгоритмов для строк с длиной от 1 до 10

Анализируя таблицу 4.1, сравним не рекурсивную реализацию алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна. Можно заметить, что на любой длине входных строк от 1 до 10 нерекурсивная реализация алгоритма поиска расстояния Левенштейна работает быстрее нерекурсивной реализации алгоритма поиска расстояния Дамерау-Левенштейна. Это связано с тем, что в последнем алгоритме добавляется еще одно условие, которое увеличивает время его работы.

При рассмотрении рекурсивной и рекурсивной с кешированием реализаций алгоритмов поиска расстояний Левенштейна можно увидеть, что, начиная со строки длиной в 3 символа, использование кеша в качестве сохранения промежуточных вычислений приводит к уменьшению времени работы почти в 2 раза, а к строке длиной в 10 символов – в 20120 раз.

Сравнивая между собой нерекурсивную и рекурсивную с кешированием реализацию алгоритма поиска расстояний Левенштейна, можно заметить, что на любой длине входных строк от 1 до 10, рекурсивная с кешированием реализация работает медленнее нерекурсивной: начиная со строки длиной в 2 символа – в 1.23 раза, а к строке длиной в 10 символов – в 1.82 раза.

Таким образом, можно сказать, что при больших длинах строк лучше использовать итеративные алгоритмы, поскольку их рост линеен, в то время как время рекурсивных алгоритмов растет экспоненциально.

4.3 Использование памяти

Введем следующие обозначения:

- l_1 — длина строки S_1;
- l_2 — длина строки S_2;
- $sizeof()$ — функция вычисляющая размер в байтах;
- $wstring$ — строковый тип;
- int — целочисленный тип;
- $size\_t$ — беззнаковый целочисленный тип.

Максимальная глубина стека вызовов при рекурсивной реализации нахождения расстояния Дамерау-Левенштейна равна сумме входящих строк, а на каждый вызов требуется 2 дополнительные переменные, соответственно, максимальный расход памяти равен:

$$(l_1 + l_2) \cdot (2 \cdot sizeof(size\_t) + 2 \cdot sizeof(int)) + 2 * sizeof(wstring), \quad (4.1)$$

где:

– $2 \cdot sizeof(size\_t)$ — хранение размеров строк;

– $2 \cdot sizeof(int)$ — дополнительные переменные;

– $2 \cdot sizeof(wstring)$ — хранение двух строк.

Расчет используемой памяти для рекурсивного алгоритма с кешированием поиска расстояния Дамерау-Левенштейна будет теоретически схож с расчетом в формуле 4.1, но также учитывается матрица, соответственно, максимальный расход памяти равен:

$$\begin{aligned}(l_1 + 1) \cdot (l_2 + 1) \cdot sizeof(int) + sizeof(int**) + (l_1 + 1) \cdot sizeof(int*) + \\ (l_1 + l_2) \cdot (2 \cdot sizeof(size\_t) + sizeof(int)) + 2 \cdot sizeof(wstring)\end{aligned} \quad (4.2)$$

где

– $(l_1 + 1) \cdot (l_2 + 1) \cdot sizeof(int)$ — хранение матрицы;

– $sizeof(int**)$ – указатель на матрицу;

– $(l_1 + 1) \cdot sizeof(int*)$ – указатель на строки матрицы;

– $2 \cdot size(size\_t)$ — хранение размеров строк;

– $sizeof(int)$ — дополнительная переменная;

– $2 * sizeof(wstring)$ — хранение двух строк.

Расчет использования памяти при итеративной реализации алгоритма поиска расстояния Левенштейна теоретически равен:

$$2 \cdot (l_2 + 1) \cdot sizeof(int) + 2 \cdot sizeof(wstring) + \\ 2 \cdot sizeof(size\_t) + sizeof(int), \tag{4.3}$$

где

– $2 \cdot (l_2 + 1) \cdot sizeof(int)$ — хранение двух массивов;

– $2 \cdot size(wstring)$ — хранение двух строк;

– $2 \cdot sizeof(size\_t)$ — хранение размеров строк;

– $sizeof(int)$ — дополнительная переменная.

Расчет использования памяти при итеративной реализации алгоритма поиска расстояния Дамерау-Левенштейна теоретически равен:

$$(l_1 + 1) \cdot (l_2 + 1) \cdot sizeof(int) + sizeof(int**) + (l_1 + 1) \cdot sizeof(int*) + \\ 2 \cdot sizeof(wstring) + 2 \cdot sizeof(size\_t) + sizeof(int), \tag{4.4}$$

где

– $(l_1 + 1) \cdot (l_2 + 1) \cdot sizeof(int)$ — хранение матрицы;

– $sizeof(int**)$ – указатель на матрицу;

– $(l_1 + 1) \cdot sizeof(int*)$ – указатель на строки матрицы;

– $2 * sizeof(wstring)$ — хранение двух строк;

– $2 \cdot size(size\_t)$ — хранение размеров матрицы;

– $sizeof(int)$ — дополнительная переменная.

Таблица 4.2 – Замер памяти для строк, размером от 10 до 200

Длина, символ	Размер, байты			
	Левенштейн	Дамерау-Левенштейн		
	Итеративный	Итеративный	Рекурсивный	
			Без кеша	С кешом
10	252	748	624	1128
20	412	2188	1184	2968
30	572	4428	1744	5608
40	732	7468	2304	9048
50	892	11308	2864	13288
60	1052	15948	3424	18328
70	1212	21388	3984	24168
80	1372	27628	4544	30808
90	1532	34668	5104	38248
100	1692	42508	5664	46488

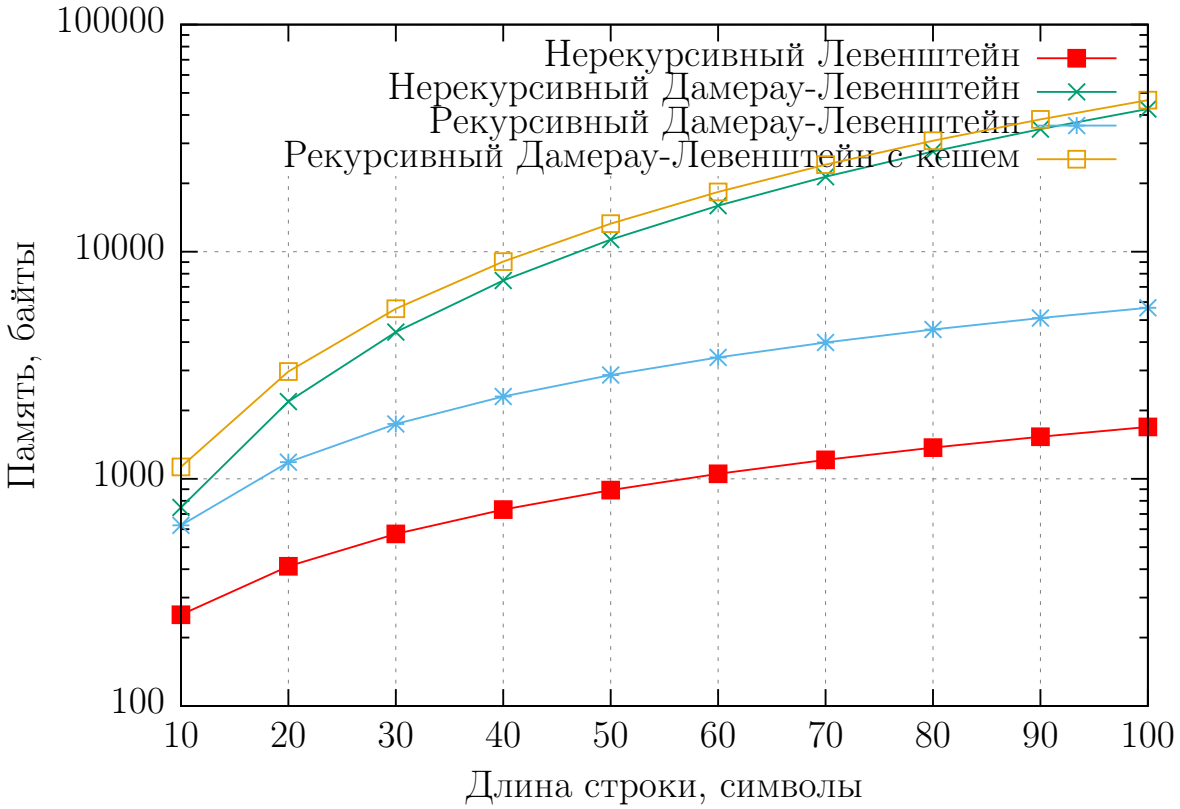

Рисунок 4.2 – Результаты вычислений используемой памяти реализаций алгоритмов для строк с длиной от 10 до 100

Анализируя таблицу 4.2, сравним нерекурсивные реализации алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна. При любой длине строки от 10 до 100 символов итеративный алгоритм поиска расстояния Левенштейна использует меньше памяти: при длине строки в 10 символов - в 3 раза меньше, а при длине строки в 100 символов уже в 25 раз. Такие результаты объясняются тем, что нерекурсивная реализация алгоритма поиска расстояния Дамерау-Левенштейна использует матрицу для сохранения ранее вычисленных значений, в то время как нерекурсивной реализации алгоритма поиска расстояния Левенштейна необходимы лишь текущая и предыдущая строки матрицы.

Сравнивая рекурсивную и рекурсивную с кешированием реализации алгоритмов поиска расстояний Левенштейна, можно увидеть, что использования матрицы в качестве кеша также приводит к быстрому росту используемой памяти в зависимости от длины входных строк.

При рассмотрении нерекурсивной и рекурсивной реализаций алгоритма поиска расстояний Дамерау-Левенштейна видно, что последняя с ростом длины строки использует в несколько раз меньше памяти, чем нерекурсивная: при длине строки в 10 символов – в 1.2 раза меньше, а при длине строки в 100 символов – в 7.5 раз меньше. Это связано с тем, что нерекурсивная реализация использует матрицу, в то время как рекурсивная использует только память, выделенную под локальные переменные при каждом рекурсивном вызове функции.

Вывод

В данном разделе были проведены замеры времени работы а также расчеты используемой памяти реализаций алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.

Итеративные реализации алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна работают быстрее рекурсивных, поскольку при итеративных реализациях не происходит многократного расчета одних и тех же промежуточных значений в ходе работы алгоритма.

Однако, рекурсивные алгоритмы более эффективные при использовании памяти, поскольку при использовании рекурсивной реализации происходит выделение памяти только под локальные переменные при каждом рекурсивном

вызове.

Использование матрицы в качестве кеша в рекурсивной реализации алгоритма Дамерау-Левенштейна позволило сократить время работы алгоритма, но увеличило количество используемой памяти.

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы были решены следующие задачи:

1) Изучены и описаны алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна;

2) Создано программное обеспечение, реализующее следующие алгоритмы:

   — нерекурсивный алгоритм поиска расстояния Левенштейна;

   — нерекурсивный алгоритм поиска расстояния Дамерау-Левенштейна;

   — рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна;

   — рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна с кешированием.

3) Проведены анализы эффективности реализаций алгоритмов по памяти и по времени;

4) Подготовлен отчет о лабораторной работе.

Цель данной лабораторной работы, а именно изучение, описание, реализация и исследование алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна, также была достигнута.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Fine-tuning and aligning question answering models for complex information extraction tasks [Электронный ресурс]. — Режим доступа: https://arxiv.org/pdf/2309.14805.pdf (дата обращения: 09.10.2022).
2. *Черненький В. М. Г. Ю. Е.* Методика идентификации пассажира по установочным данным //. Т. 163. — – М.: Вестник МГТУ им. Н.Э. Баумана. Сер. “Приборостроение”, 2012. — Гл. 4. С. 30—34.
3. Справочник по языку C++ [Электронный ресурс]. — Режим доступа: https://learn.microsoft.com/ru-ru/cpp/cpp/cpp-language-reference?view=msvc-170 (дата обращения: 28.09.2022).
4. clock_getres [Электронный ресурс]. — Режим доступа: https://pubs.opengroup.org/onlinepubs/9699919799/functions/clock_getres.html (дата обращения: 28.09.2022).
5. Ubuntu 22.04.3 LTS (Jammy Jellyfish) [Электронный ресурс]. — Режим доступа: https://releases.ubuntu.com/22.04/ (дата обращения: 28.09.2022).