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

по курсу «Анализ Алгоритмов»

на тему: «Динамическое программирование»

Студент ИУ7-56Б _____ Мансуров В. М.
(Группа) (Подпись, дата) (И. О. Фамилия)

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

2022 г.

СОДЕРЖАНИЕ

Введение

В данной лабораторной работе будет рассмотрено расстояние Левенштейна. Данное расстояние показывает минимальное количество операций (вставка, удаление, замены), которое необходимо для перевода одной строки в другую. Это расстояние помогает определить схожесть двух строк.

Впервые задачу поставил в 1965 году советский математик Владимир Левенштейн при изучении последовательностей 0-1, впоследствии более общую задачу для произвольного алфавита связали с его именем.

Расстояние Левенштейна применяется в теории информации и компьютерной лингвистике 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;
- для сравнения генов, хромосом и белков в биоинформатике.

Целью данной лабораторной работы является изучение и исследование особенностей задач динамического программирования на алгоритмах Левенштейна и Дамерау-Левенштейна.

Для поставленной цели необходимо выполнить следующие задачи:

1) изучить расстояния Левенштейна и Дамерау-Левенштейна;
2) создать ПО, реализующее следующие алгоритмы:
    - нерекурсивный метод поиска расстояния Левенштейна;
    - нерекурсивный метод поиска Дамерау-Левенштейна;
    - рекурсивный метод поиска Дамерау-Левенштейна;
    - рекурсивный с кешированием метод поиска Дамерау-Левенштейна.
3) выбрать инструменты для замера процессорного времени выполнения реализаций алгоритмов;
4) Провести анализ затрат работы программы по времени и по памяти, выяснить влияющие на них характеристики.

1 Аналитическая часть

1.1 Расстояние Левенштейна

Расстояние Левенштейна (редакционное расстояние, дистанция редактирования) — метрика, измеряющая разность между двумя последовательностями символов.
Цены операций могут зависеть от вида операций:

1. $w(a, b)$ — цена замены символа a на b, R (от англ. replace);
2. $w(\lambda, b)$ — цена вставки символа b, I (от англ. insert);
3. $w(a, \lambda)$ — цена удаления символа a, D (от англ. delete).

Будем считать стоимость каждой вышеизложенной операции равной 1, то есть:

- $w(a, b) = 1$, $a \neq b$;
- $w(\lambda, b) = 1$;
- $w(a, \lambda) = 1$.

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

Расстояние Левенштейна между двумя строками S_1 и S_2, длинной M и N соответственно. Расстояние Левенштейна рассчитывается по рекуррентной формуле **??**:

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

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

1.1.1 Рекурсивный алгоритм нахождения расстояние Левенштейна

Рекурсивный алгоритм реализует формулу 1.1, функция D составлена таким образом, что:

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(удаление, вставка, замена) в некоторой последовательности. Последовательность поведения любых двух операций можно поменять, порядок поведения операций не имеет никакого значения. Пологая, что a', b' - строки a и b без последнего символа соответственно, цена преобразования из строки a в строку 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 разные символы;
    - цена преобразования из a' в b' , предполагая, что a и b оканчиваются на один и тот же символ.

Минимальной ценой преобразования будет минимальное значение приведенных вариантов.

1.1.2 Рекурсивный алгоритм нахождения расстояния Левенштейна с кэшированием

Рекурсивная реализация алгоритма Левенштейна малоэффективна по времени при больших i, j, по причине проблемы повторных вычислений $D(i, j)$. Для оптимизации нахождения расстояния Левенштейна можно использовать матрицу в целях хранения соответствующих промежуточных значений. В таком случае алгоритм представляет собой рекурсивное заполнение матрицы $A_{|a|,|b|}$ значениями $D(i, j)$, такое хранение промежуточных данных можно назвать кэшом для рекурсивного алгоритма.

1.1.3 Нерекурсивный алгоритм нахождения расстояния Левенштейна

Рекурсивная реализация алгоритма Левенштейна с кэшированием малоэффективна по времени при больших i, j. Для оптимизации можно использовать итерационную реализацию заполнение матрицы промежуточными значениями $D(i, j)$.

Однако матричный алгоритм является малоэффективным по памяти при больших i, j, т.к. множество промежуточных значений $D(i, j)$ хранится в памяти после их использования. Для оптимизации нахождения расстояния Левенштейна можно использовать кэш, т.е. пару строк, содержащюю значения $D(i, j)$, вычисленные в предыдущей итерации алгоритма и значения $D(i, j)$, вычисляемый в текущей итерации.

1.2 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна(названо в честь учёных Фредерика Дамерау и Владимир Левенштейна) - это мера разницы двух строк символов, определяемая как минимальное количество операций вставки, удаления, замены и транспонзиций (перестановки двух соседних символов), необходимых для перевода одной строки в другую. Является модификацией расстояния Левенштейна, то есть к операциям добавляется операция транпонзиция T (от англ. transposition).

Расстояние Дамерау-Левенштейна может быть вычисленно по рукуррет-ной формуле:

$$D(i,j) = \begin{cases} 0 & ,j = 0, i = 0 \\ i & ,j = 0, i > 0 \\ j & ,j > 0, i = 0 \\ min \begin{cases} D(i,j-1)+1 \\ D(i-1,j)+1 \\ D(i-1,j-1)+m(S_1[i],S_2[i]) \\ D(i-2,j-2)+m(S_1[i],S_2[i]) \end{cases} & \begin{array}{l} \text{, если i} > 1\text{, j} > 1 \\ ,S_1[i] = S_2[j-1] \\ ,S_1[j] = S_2[i-1] \end{array} \\ min \begin{cases} D(i,j-1)+1 \\ D(i-1,j)+1 \\ D(i-1,j-1)+m(S_1[i],S_2[i]) \end{cases} & \text{,иначе} \end{cases} \quad (1.3)$$

1.3 Вывод

В данном разделе были рассмотрены алгоритмы динамического программирования - алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна, формулы которых задаются реккурентно, а следовательно, данные алгоритмы могут быть реализованы рекурсивно и итерационно. На вход алгоритмам поступают две строки, которые могут содержать как русские, так и английские буквы, также будет предусмотрен ввод пустых строк.

2 Конструкторская часть

В данном разделе будут приведены схемы алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна, приведено описанию используемых типов данных, оценки памяти, а также описана структура ПО.

2.1 Разработка алгоритмов

На вход алгоритмов подаются строки S_1 и S_2

На рисунке 2.1 представлен схема алгоритма поиска расстояния Левенштейна.

На рисунках 2.2 - 2.5 представлены схемы алгоритмов поиска Дамерау-Левенштейна.

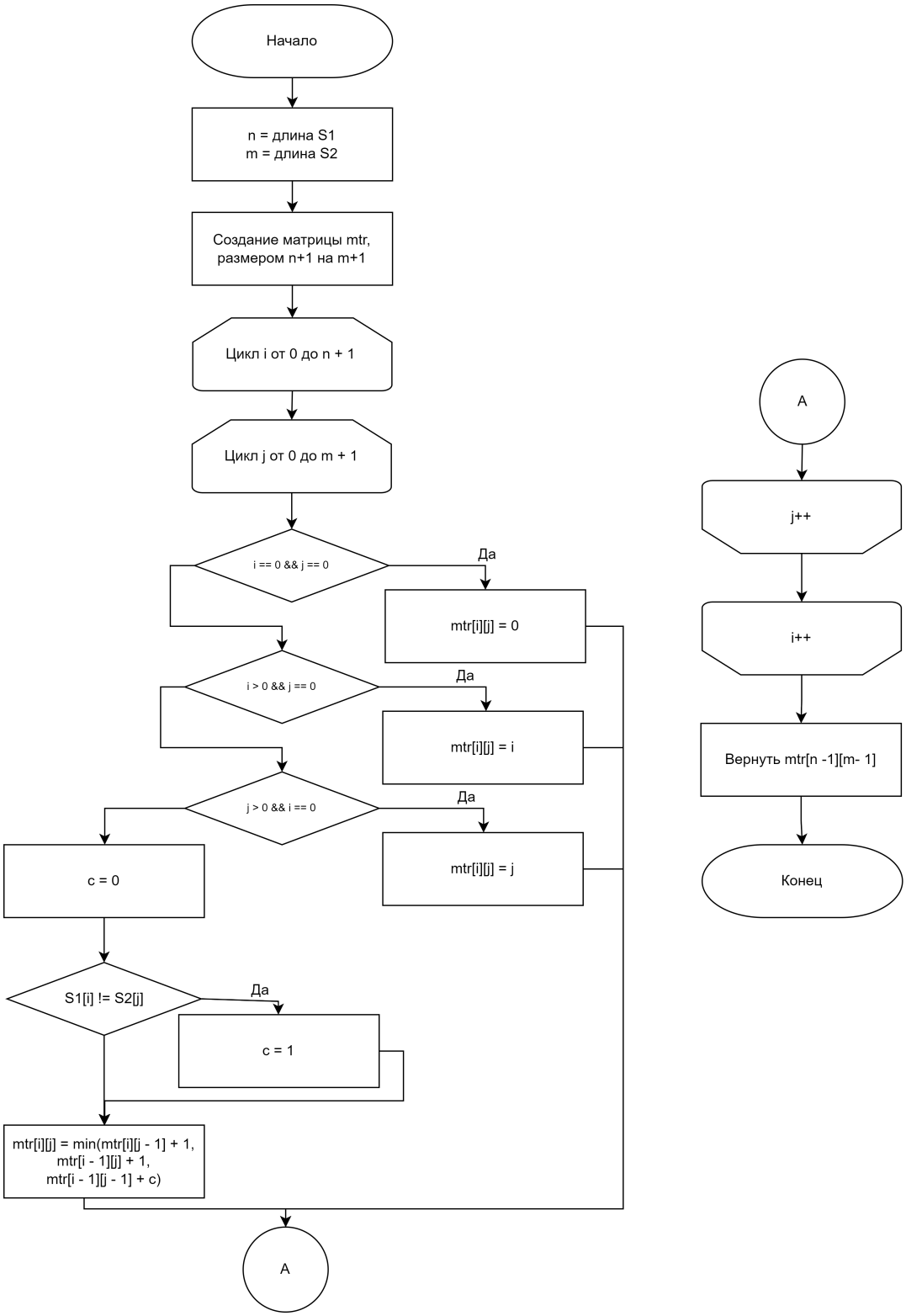

Рисунок 2.1 – Схема нерекурсивного алгоритма нахождения расстояния Левенштейна

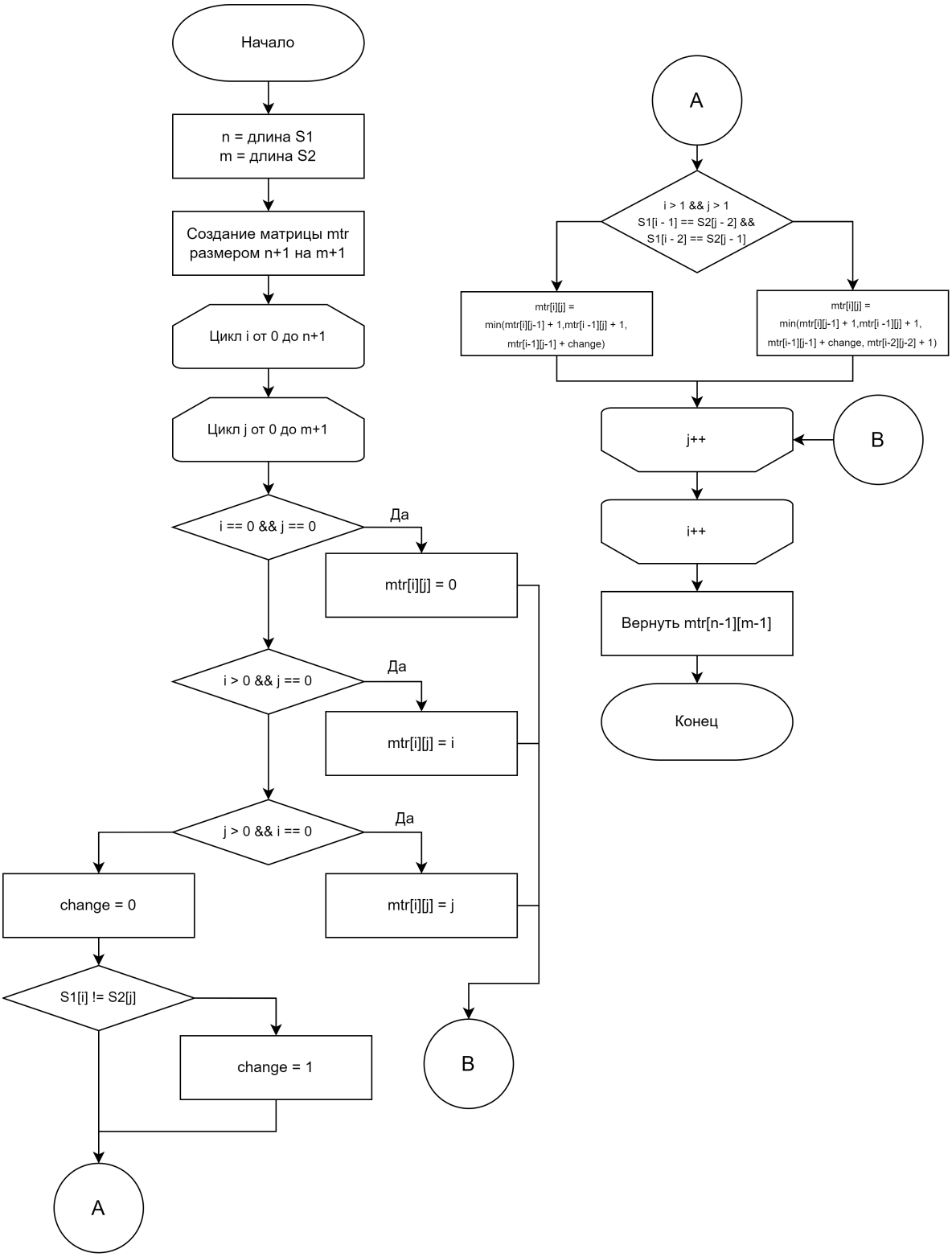

Рисунок 2.2 – Схема нерекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

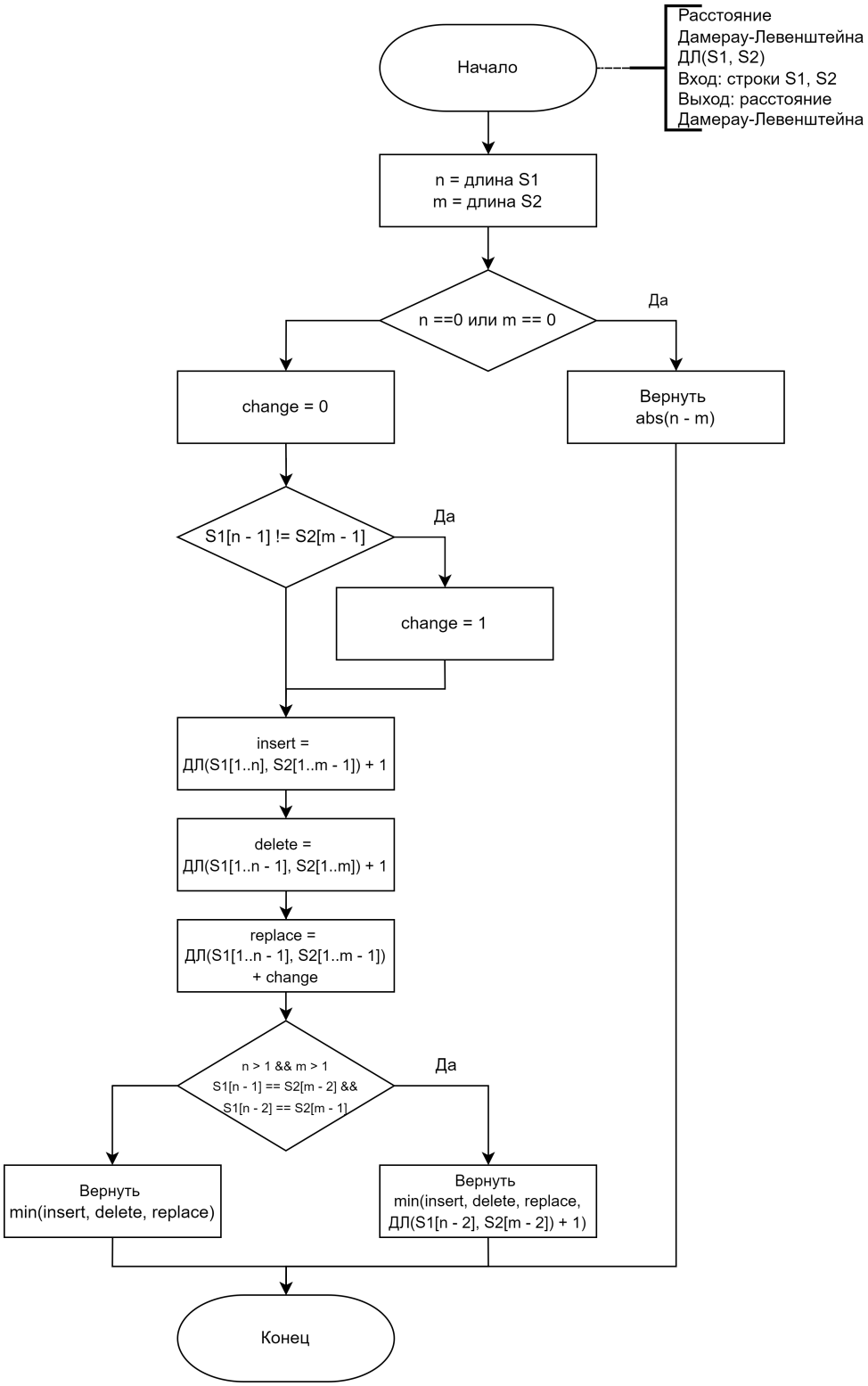

Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

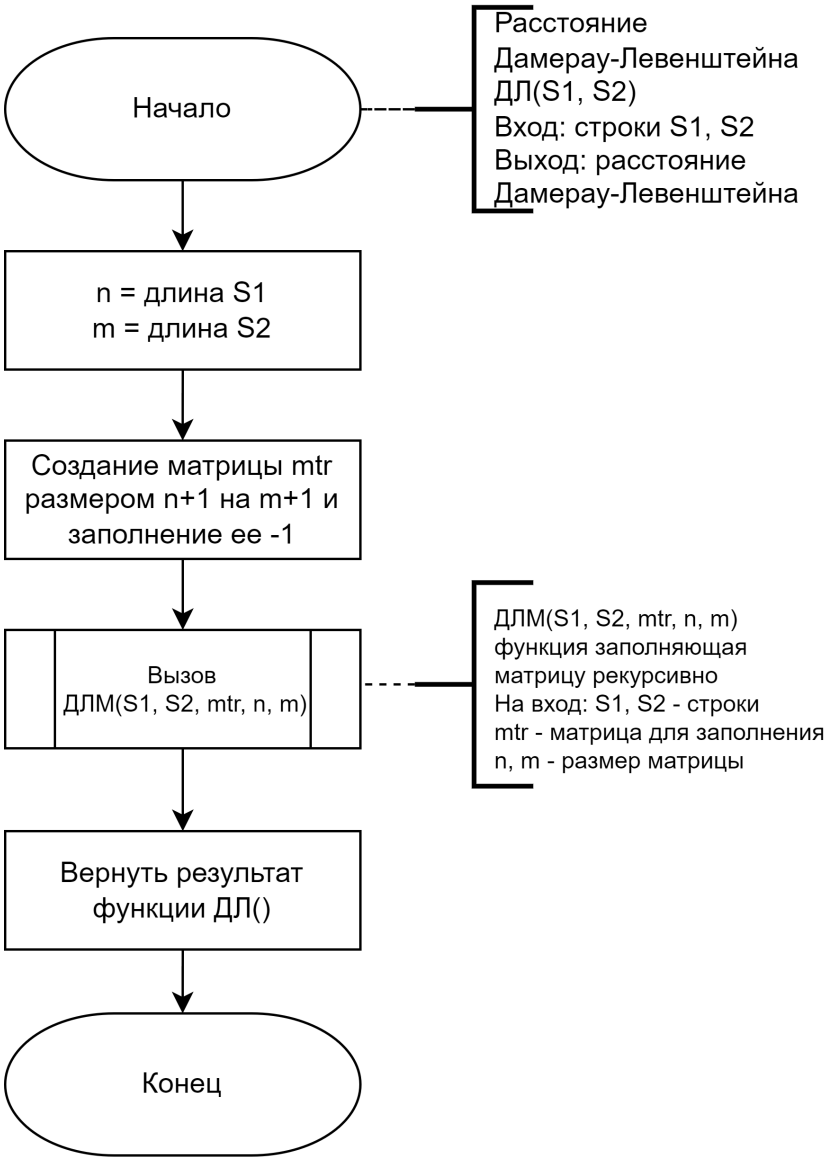

Рисунок 2.4 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с кэшированием

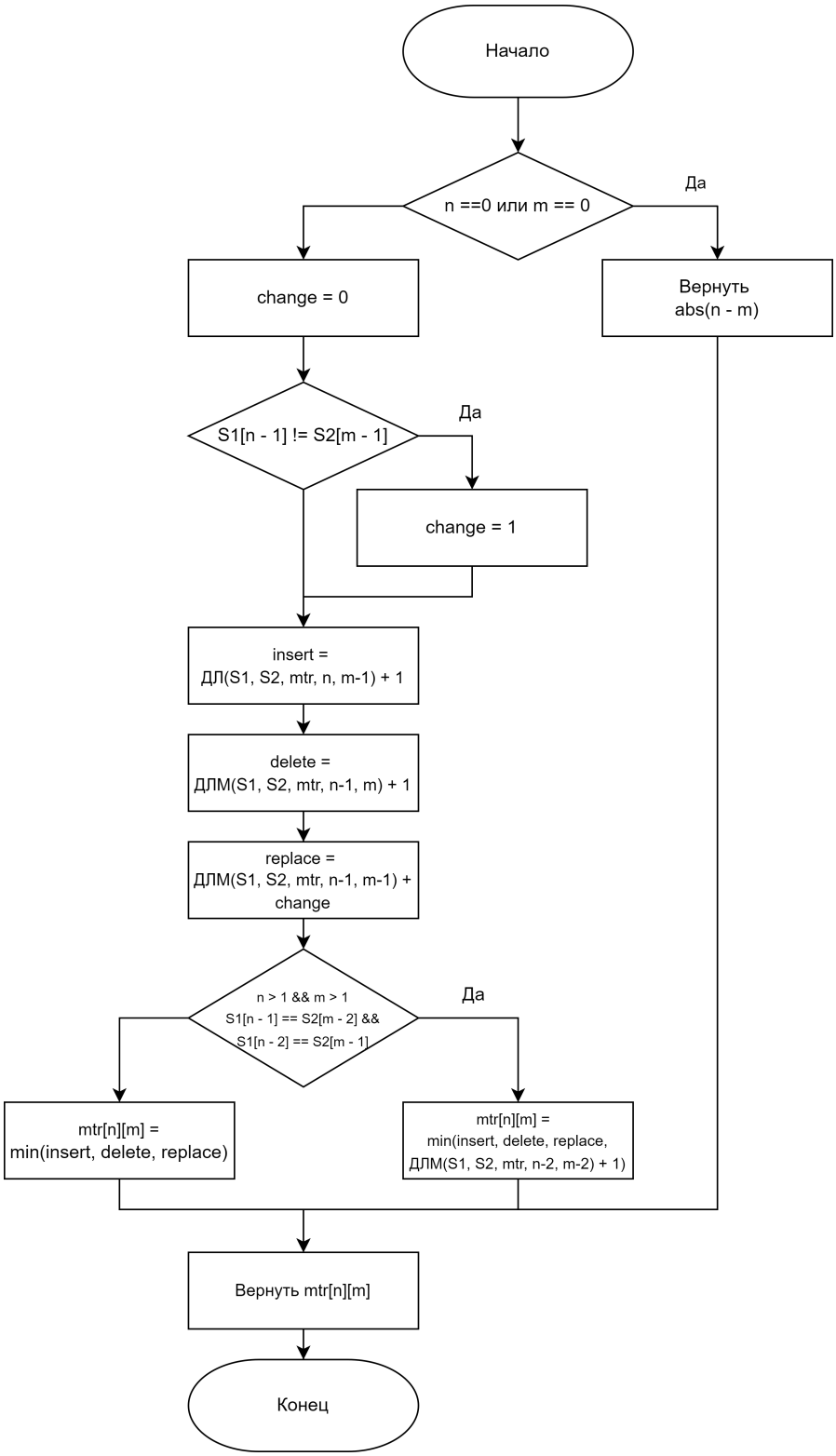

Рисунок 2.5 – Схема алгоритма рекурсивного заполнения матрицы расстоянием Дамерау-Левенштейна

2.2 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие структуры данных:

- строка - массив типа *char* размером длины строки;
- длина строки - целое число типа *int*;
- матрица - двумерный массив типа *int*.

2.3 Вывод

В данном разделе на основе теоретических данных были построены схемы требуемых алгоритмов, выбраны используемые типы данных.

3 Технологическая часть

В данном разделе будут приведены требования к программному обеспечению, средства реализации, листинг кода и функциональные тесты.

3.1 Требования к программному обеспечению

К программе предъявлены ряд требований:

- входные данные - две строки на русском или английском языке в любом регистре;
- На выходе — результат выполнения каждого из вышеуказанных алгоритмов.

3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язык $C++$. Данный выбор обусловлен тем, что я имею некоторый опыт разработки на нем, а так же наличием у языка встроены библиотеки измерения процессорного времени и тип данных работающий как с кириллицей, так и с латиницей – $std :: wstring$.

3.3 Сведения о модулях программы

Данная программа разбита на следующие модули:

- main.cpp – Файл, содержащий точку входа в программу. В нем происходит общение с пользователем и вызов алгоритмов;
- algorithms.cpp — Файл содержит функции поиска расстояния Левенштейна и Дамерау-Левенштейна.
- allocate.cpp — Файл содержит функции динамического выделения и очищения памяти для матрицы.
- print_mtr_lev.cpp – Файл содержит функции вывода матрицы для итерационных методов поиска расстояния Левенштейна и Дамеру-Левештейна, включая строки.

- cpu_time.cpp — Файл содержит функции, замеряющее процессорное время алгоритмов поиска расстояния Левеншьейна и Дамерау-Левенштейна.
- memory.cpp — Файл содержит функции, замеряющее память итерационного и рекурсивного алгоритмов поиска расстояния Левенштейна.

Листинг 3.1 – Функция нахождения расстояния Левенштейна с использованием матрицы

```
1  int lev_mtr(wstring &str1, wstring &str2, bool print)
2  {
3      size_t n = str1.length();
4      size_t m = str2.length();
5      int **mtr = malloc_mtr(n + 1, m + 1);
6      int res = 0;
7
8      for (int i = 0; i <= n; i++)
9          for (int j = 0; j <= m; j++)
10             if (i == 0 && j == 0)
11                 mtr[i][j] = 0;
12             else if (i > 0 && j == 0)
13                 mtr[i][j] = i;
14             else if (j > 0 && i == 0)
15                 mtr[i][j] = j;
16             else {
17                 int change = 0;
18                 if (str1[i - 1] != str2[j - 1])
19                     change = 1;
20
21                 mtr[i][j] = std::min(mtr[i][j - 1] + 1,
22                                      std::min(mtr[i - 1][j] + 1,
23                                               mtr[i - 1][j - 1] +
                                                    change));
24             }
25
26     if (print)
27         print_mtr_lev(str1, str2, mtr, n, m);
28     res = mtr[n][m];
29     free_mtr(mtr, n);
30
31     return res;
32 }
```

Листинг 3.2 – Функция нахождения расстояния Дамерау-Левенштейна с использованием матрицы

```
1  int dameray_lev_mtr(wstring &str1, wstring &str2, bool print)
2  {
3      size_t n = str1.length();
4      size_t m = str2.length();
5      int **mtr = malloc_mtr(n + 1, m + 1);
6      int res = 0;
7
8      for (int i = 0; i <= n; i++)
9          for (int j = 0; j <= m; j++) {
10             if (i == 0 && j == 0)
11                 mtr[i][j] = 0;
12             else if (i > 0 && j == 0)
13                 mtr[i][j] = i;
14             else if (j > 0 && i == 0)
15                 mtr[i][j] = j;
16             else {
17                 int change = 0;
18                 if (str1[i - 1] != str2[j - 1])
19                     change = 1;
20
21                 mtr[i][j] = min(mtr[i][j - 1] + 1,
22                                 min(mtr[i - 1][j] + 1,
23                                     mtr[i - 1][j - 1] + change));
24
25                 if (i > 1 && j > 1 &&
26                     str1[i - 1] == str2[j - 2] &&
27                     str1[i - 2] == str2[j - 1])
28                     mtr[i][j] = min(mtr[i][j], mtr[i - 2][j - 2] +
                           1);
29             }
30         }
31
32     if (print)
33         print_mtr_lev(str1, str2, mtr, n, m);
34     res = mtr[n][m];
35     free_mtr(mtr, n);
36
37     return res;
38 }
```

Листинг 3.3 – Функция нахождения расстояния Дамерау-Левенштейна рекурсивно

```
1  int dameray_lev_rec_t(wstring &str1, wstring &str2, size_t n,
       size_t m) {
2      if (n == 0)
3          return m;
4      if (m == 0)
5          return n;
6
7      int change = 0;
8      int res = 0;
9      if (str1[n - 1] != str2[m - 1])
10         change = 1;
11
12     res = min(dameray_lev_rec_t(str1, str2, n, m - 1) + 1,
13               min(dameray_lev_rec_t(str1, str2, n - 1, m) + 1,
14                   dameray_lev_rec_t(str1, str2, n - 1, m - 1) +
                        change));
15
16     if (n > 1 && m > 1 &&
17         str1[n - 1] == str2[m - 2] &&
18         str1[n - 2] == str2[m - 1])
19         res = std::min(res, dameray_lev_rec_t(str1, str2, n - 2, m
              - 2) + 1);
20     return res;
21 }
22
23 int dameray_lev_rec(wstring &str1, wstring &str2)
24 {
25     return dameray_lev_rec_t(str1, str2, str1.length(),
          str2.length());
26 }
```

Листинг 3.4 – Функция нахождения расстояния Дамерау-Левенштейна рекурсивно с кэшированием

```
1  int dameray_lev_rec_hash_t(wstring &str1, wstring &str2, int **mtr,
       size_t n, size_t m)
2  {
3      if (n == 0)
4          return mtr[n][m] = m;
5      if (m == 0)
6          return mtr[n][m] = n;
7      int change = 0;
8      if (str1[n - 1] != str2[m - 1])
9          change = 1;
10     mtr[n][m] = min(dameray_lev_rec_hash_t(str1, str2, mtr, n, m -
           1) + 1,
11                     min(dameray_lev_rec_hash_t(str1, str2, mtr, n -
                           1, m) + 1,
12                         dameray_lev_rec_hash_t(str1, str2, mtr, n -
                               1, m - 1) + change));
13     if (n > 1 && m > 1 &&
14         str1[n - 1] == str2[m - 2] &&
15         str1[n - 2] == str2[m - 1])
16         mtr[n][m] = min(mtr[n][m], dameray_lev_rec_hash_t(str1,
               str2, mtr, n - 2, m - 2) + 1);
17     return mtr[n][m];
18 }
19
20 int dameray_lev_rec_hash(wstring &str1, wstring &str2, bool print)
21 {
22     size_t n = str1.length();
23     size_t m = str2.length();
24     int **mtr = malloc_mtr(n + 1, m + 1);
25     for (int i = 0; i <= n; i++)
26         for (int j = 0; j <= m; j++) {
27                 mtr[i][j] = -1;
28         }
29     int res = dameray_lev_rec_hash_t(str1, str2, mtr, n, m);
30     if (print)
31         print_mtr_lev(str1, str2, mtr, n, m);
32     free_mtr(mtr, n);
33     return res;
34 }
```

Листинг 3.5 – Функции динамического выделения и очищения памяти под матрицу

```
1  void free_mtr(int **mtr, std::size_t n) {
2      if (mtr != nullptr)
3      {
4          for (std::size_t i = 0; i < n; i++)
5              if (mtr[i] != nullptr)
6                  free(mtr[i]);
7          free(mtr);
8      }
9  }
10
11 int **malloc_mtr(std::size_t n, std::size_t m)
12 {
13     if (n == 0)
14         return nullptr;
15
16     int **mtr = static_cast<int **>(malloc(n * sizeof(int *)));
17     if (mtr != nullptr)
18         for (std::size_t i = 0; mtr[i] != nullptr && i < n; i++) {
19             mtr[i] = static_cast<int *>(malloc(m * sizeof(int)));
20             if (mtr[i] == nullptr)
21                 free_mtr(mtr, n);
22         }
23
24     return mtr;
25 }
```

Листинг 3.6 – Функции вывода матрицы для алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна

```
void print_mtr_lev(std::wstring str1, std::wstring str2,
                   int **mtr, std::size_t n, std::size_t m)
{
    for(std::size_t i = 0; i <= n + 1; i++)
    {
        for(std::size_t j = 0; j <= m + 1; j++)
        {
            if (i == 0 && j == 0)
                std::wcout << "  ";
            else if (i == 0)
                if (j == 1)
                    std::wcout << "- ";
                else
                    std::wcout << str2[j - 2] << " ";
            else if (j == 0)
                if (i == 1)
                    std::wcout << "- ";
                else
                    std::wcout << str1[i - 2] << " ";
            else
                std::wcout << mtr[i - 1][j - 1] << " ";
        }
        std::wcout << std::endl;
    }
}
```

3.4 Функциональные тесты

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

Таблица 3.1 – Функциональные тесты

Строка 1	**Строка 2**	**Левенштейн**	**Дамерау-Левенштейн**		
		Итеративный	**Итеративный**	**Рекурсивный**	
				Без кэша	**С кэшом**
a	b	1	1	1	1
a	a	0	0	0	0
кот	скат	2	2	2	2
друзья	рдузия	3	2	2	2
вагон	гонки	4	4	4	4
бар	раб	2	2	2	2
слон	слоны	1	1	1	1

3.5 Вывод

Были реализованы алгоритмы: вычисления расстояния Левенштейна итерационно, а также вычисления расстояния Дамерау– Левенштейна итерауионно, рекурсивно и вычисления расстояния Дамерау–Левенштейна с рекурсивного заполнением кэша. Проведено тестирование разработанных алгоритмов.

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование представлены далее.

- Процессор: Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz 2.50 GHz.
- Оперативная память: 16 GiB.
- Операционная система: Windows 10 Pro 21H2 64-bit.

При тестировании ноутбук был включен в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, а также системой тестирования.

4.2 Демонстрация работы программы

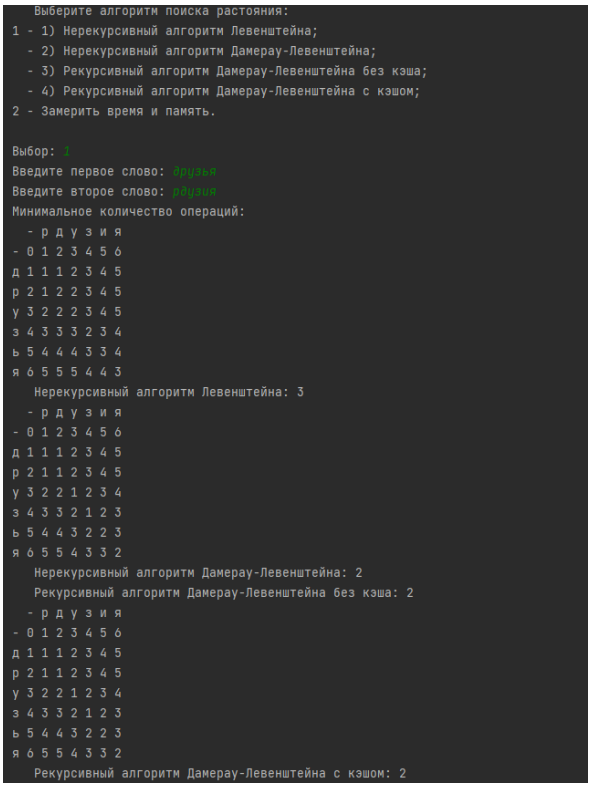

Рисунок 4.1 – Демонстрация работы программы при поиске расстояние Левенштейна и Дамерау-Левенштейна

4.3 Временные характеристики

Результаты замеров по результатам экспериментов приведены в Таблице 4.1. В данной таблице для значений, для которых тестирование не выполнялось, в поле результата находится " - ".

Таблица 4.1 – Замер времени для строк, размером от 1 до 200

Длина (символ)	**Время, нс**			
	Левенштейн	**Дамерау-Левенштейн**		
	Итеративный	**Итеративный**	**Рекурсивный**	
			Без кэша	**С кэшом**
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
5	0	15.656	15.656	15.656
6	0	0	62.625	62.625
7	15.656	15.656	328.78	391.41
8	0	0	1706.5	2004
9	15.656	15.656	9393.8	11288
10	15.656	15.656	51932	61545
20	15.656	15.656	-	-
30	31.313	15.656	-	-
40	31.313	31.313	-	-
50	31.313	46.969	-	-
60	62.625	78.282	-	-
70	78.282	78.282	-	-
80	93.938	109.59	-	-
90	109.59	140.91	-	-
100	140.91	172.22	-	-
200	673.22	751.5	-	-

Отдельно сравним итеративные алгоритмы поиска расстояний Левенштейна и Дамерау–Левенштейна. Сравнение будет производится на основе данных, представленных в Таблице 4.1. Результат можно увидеть на Рисунке 4.2.

При длинах строк менее 30 символов разница по времени между итеративными реализациями незначительна, однако при увеличении длины строки алгоритм поиска расстояния Левенштейна оказывается быстрее вплоть до полутора раз (при длинах строк равных 200). Это обосновывается тем, что у алгоритма поиска расстояния Дамерау-Левенштейна задействуется дополнительная операция, которая замедляет алгоритм

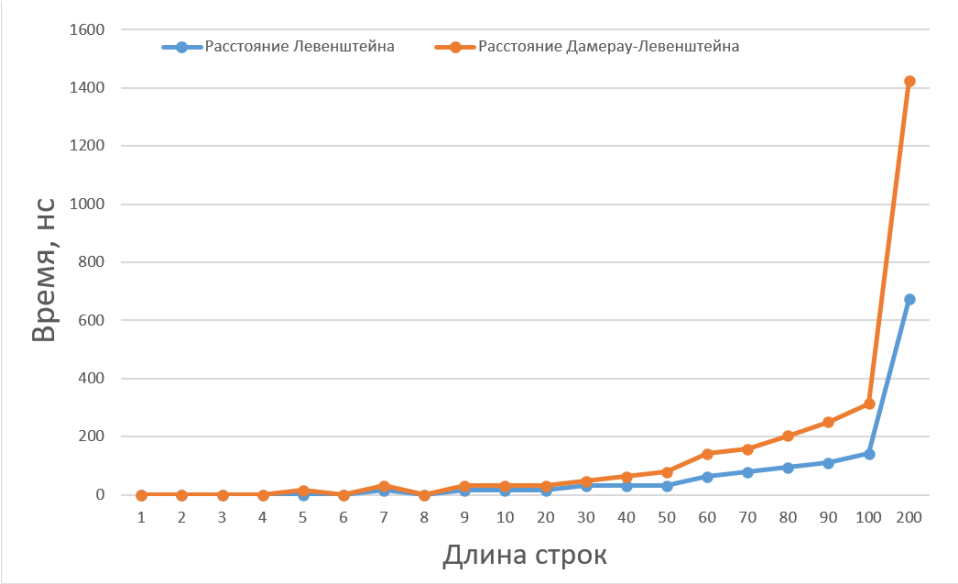

Рисунок 4.2 – Сравнение по времени алгоритмов поиска расстояния Левенштейн и Дамерау-Левенштейна – нерекурсивной реализации

Так же сравним рекурсивную и итеративную реализации алгоритма поиска расстояния Дамерау-Левенштейна. Данные представлены в Таблице 4.1 и отображены на Рисунке 4.3.

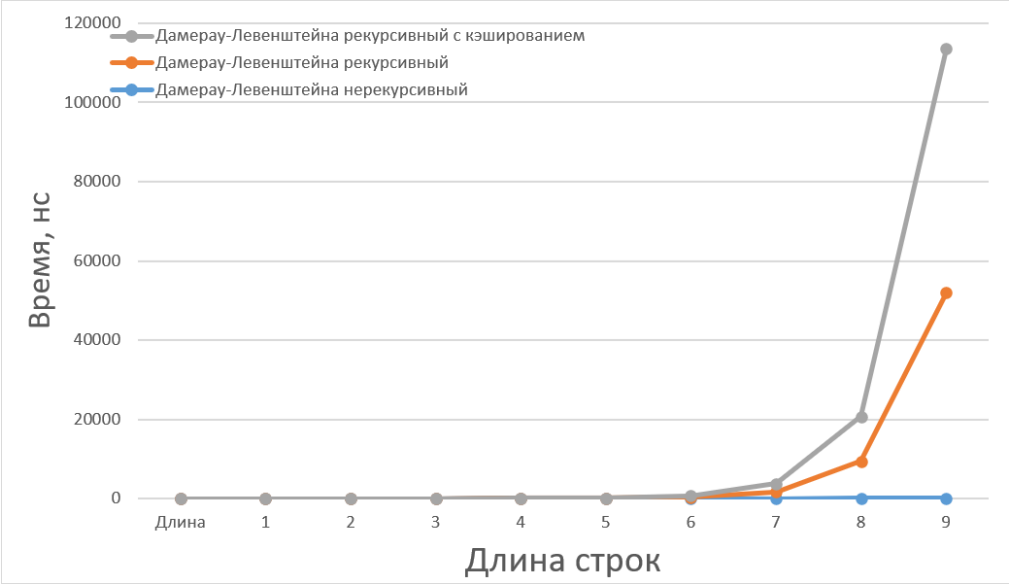

Рисунок 4.3 – Сравнение по времени алгоритмов поиска расстояния Дамерау-Левенштейна

На Рисунке 4.3 продемонстрировано, что рекурсивный алгоритм становится менее эффективным (вплоть до 21 раз при длине строк равной 7 элементов), чем итеративный.

Из этого можно сделать вывод о том, что при малых длинах строк (1-4 символа) предпочтительнее использовать рекурсивные алгоритмы, однако

при обработке более длинных строк (болеее 3 символов) итеративные алгоритмы оказываются многократно более эффективными и рекомендованы к использованию.

Из данных, приведенных в Таблице 4.1, видно, что итеративные алгоритмы становятся более эффективными по времени при увеличении длин строк, работая приблизительно в 308 млн. раз (Левенштейн) и 203 млн. раз (Дамерау-Левенштейн) быстрее, чем рекурсивные (при длинах строк равных 200). Однако, при малых длинах (1-4 элемента) рекурсивные алгоритмы работает по примерно как итеративные.

Кроме того, согласно данным, приведенным в Таблице 4.1, рекурсивные алгоритмы при длинах строк более 10 элементов не пригодны к использованию в силу экспоненциально роста затрат процессорного времени, в то время, как затраты итеративных алгоритмов по времени линейны.

4.4 Характеристики по памяти

Алгоритмы Левенштейна и Дамерау-Левенштейна не отличаются по использованию памяти, поэтому достаточно рассмотреть будет рассмотреть рекурсивную и матричную реализацию одного из этих алгоритмов.

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме входящих строк, а на каждый вызов требуется 3 дополнительных переменных типа $integer$, соответственно, максимальный расход памяти

$$(Len(S_1) + Len(S_2)) \cdot (2 \cdot Size(\text{string}) + 5 \cdot Size(\text{int})), \tag{4.1}$$

где S_1, S_2 - строки, Size - функция, возвращающая размер аргумента; Len - функция, возвращающая длину строки, string - строковый тип, int - целочисленный.

Использование памяти при итеративной реализации теоретически равно:

$$(Len(S_1) + 1) \cdot (Len(S_2) + 1) \cdot Size(int) + 3 \cdot Size(int) + 2 \cdot Size(string) \tag{4.2}$$

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

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

	Размер, байт	
Длина (символ)	**Итеративный**	**Рекурсивный**
10	476	1680
20	1676	3360
30	3676	5040
40	6476	6720
50	10076	8400
60	14476	10080
70	19676	11760
80	25676	13440
90	32476	15120
100	40076	16800
110	48476	18480
120	57676	20160
130	67676	21840
140	78476	23520
150	90076	25200
160	102476	26880
170	115676	28560
180	129676	30240
190	144476	31920
200	160076	33600

Из данных, приведенных в Таблице 4.2, видно, что рекурсивные алгоритмы являются более эффективными по памяти, так как используется только память под локальные переменные, передаваемые аргументы и возвращаемое значение, в то время как итеративные алгоритмы затрачивают память линейно пропорционально длинам обрабатываемых строк.

В связи с этим, при недостаточном объеме памяти, рекомендуются использовать рекурсивные алгоритмы, так как они не используют дополнительной памяти в процессе работы.

На Рисунке 4.4 продемонстрировано, что итерационный алгоритм становится менее эффективным, чем рекурсивный если длина строк больше 40, достаточно хорошо это продемонстрировано на Рисунке 4.5.

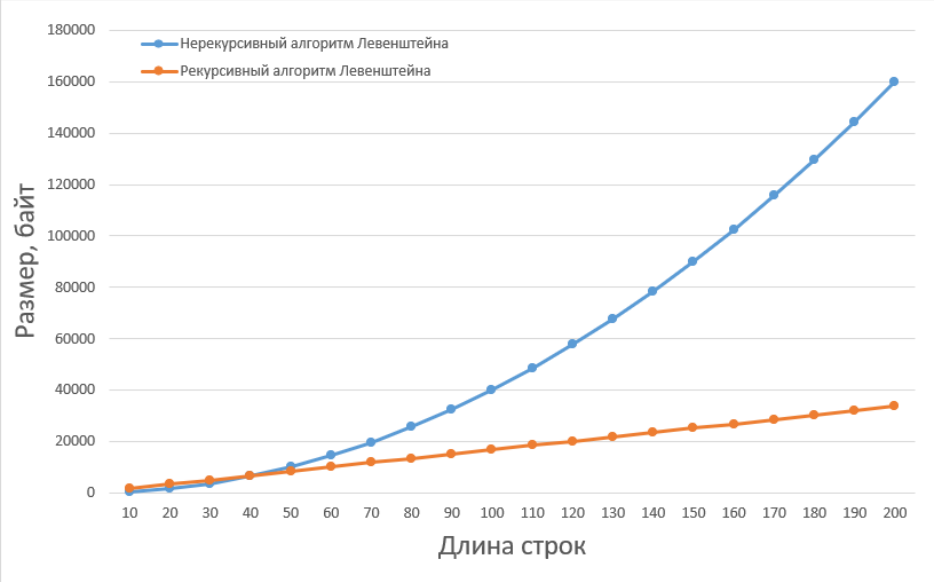

Рисунок 4.4 – Сравнение по памяти алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна – итеративной и рекурсивной реализации

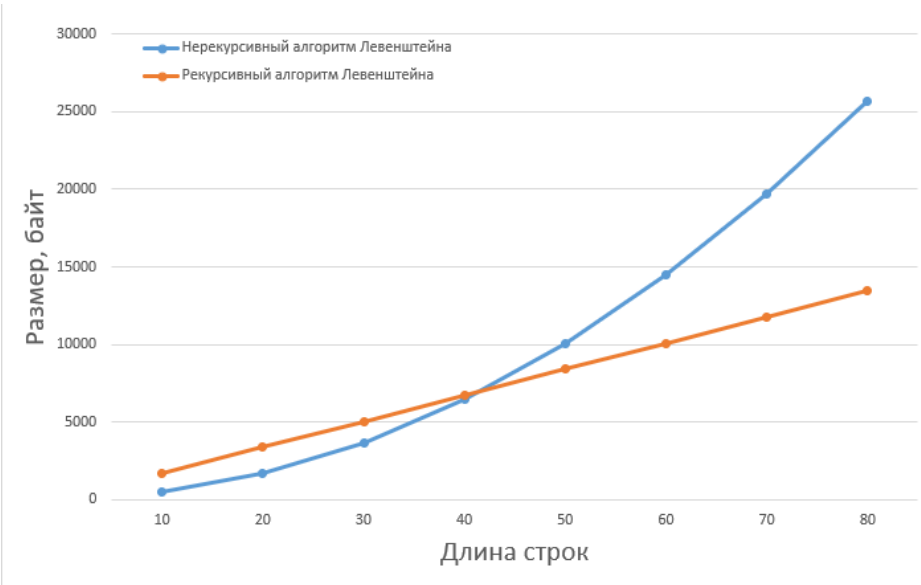

Рисунок 4.5 – Сравнение по памяти алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна – итеративной и рекурсивной реализации

4.5 Сравнительный анализ алгоритмов

Приведенные характеристики показывают нам, что рекурсивная реализация алгоритма очень сильно проигрывает по времени. В связи с этим, рекурсивные алгоритмы следует использовать лишь для малых размерностей строк (1-4 символа) или при малом объеме оперативной памяти.

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 проигрывает по времени и памяти алгоритму Левенштейна.

Можно сделать вывод о том, что рекуррентный алгоритм поиска расстояния Дамерау-Левенштейна будет более затратным по времени по сравнению с итеративной реализацией алгоритма поиска расстояния Дамерау-Левенштейна.

4.6 Вывод

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

Для обработок малых длин строк (1-4 символа) предпочтительнее использовать рекурсивные алгоритмы, в то время как для остальных случаев рекомендуются использовать итеративные реализации. Однако, стоит учитывать дополнительные затраты по памяти, возникающие при использовании итеративных алгоритмов.

Заключение

Алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна являются самыми популярными алгоритмами, которые помогают найти редакторское расстояние.

Поставленные цели работы были достигнуты и выполнены, т.е. в результате выполнения данной лабораторной работы были изучены алгоритмы поиска расстояний Левенштейна (Формула 1.1) и Дамерау-Левенштейна (Формула 1.3), построены схемы (Рисунок 2.1, Рисунок **??**), соответствующие данным алгоритмам, также разобраны рекурсивные алгоритмы (Рисунок 2.3, Рисунок 2.4). Реализован программный продукт, который вычисляет дистанцию 4 способами.

В рамках выполнения работы решены следующие задачи.

- Изучены расстояния Левенштейна и Дамерау–Левенштейна.
- Реализованы алгоритмы поиска расстояний:
  - Нерекурсивный алгоритм нахождения расстояния Левенштейна.
  - Нерекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна.
  - Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна.
  - Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна с кэшированием.
- Замерено процессорное время работы реализаций алгоритмов поиска расстояний.
- Проведено сравнение временных характеристик, а также затраченной памяти.

Список литературы

[1] Левенштейн В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов. – М.: Доклады АН СССР, 1965. Т. 163. С. 845– 848. Режим доступа: https://goo.su/8xIsL

[2] Документация по Microsoft C++ [Электронный ресурс]. Режим доступа: https://learn.microsoft.com/ru-ru/cpp/?view=msvc-170&viewFallbackFrom=vs-2017 (дата обращения: 25.09.2022).

[3] time — Time access and conversions [Электронный ресурс]. Режим доступа: https://www.tutorialspoint.com/c_standard_library/c_function_clock.htm (дата обращения: 25.09.2022).

[4] Windows 10 Pro 2h21 64-bit Электронный ресурс]. Режим доступа: https://www.microsoft.com/ru-ru/software-download/windows10 (дата обращения: 25.09.2022).

[5] Intel [Электронный ресурс]. Режим доступа: https://ark.intel.com/content/www/ru/ru/ark/products/201839/intel-core-i510300h-processor-8m-cache-up-to-4-50-ghz.html (дата обращения: 25.09.2022).