СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Пусть дано — строка $S1$, длина которой $L1$ и строка $S2$, длина которой $L2$. Тогда **расстояние Левенштейна** это минимальное количество редакторных операций вставки, удаления и замены символов, необходимых для превращения одной строки в другую.

Расстояние Дамерау-Левенштейна является расширением расстояния Левенштейна, которое включает дополнительную операцию - транспозицию, чтобы обработать случаи, когда символы меняются местами или переупорядочиваются.

Расстояния Левенштейна и Дамерау-Левенштейна используются при решении следующих задач:

— компьютерная лингвистика;

— биоинформатика;

— корректировка поискового запроса.

Целью данной лабораторной работы является изучение алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.

Необходимо выполнить следующие **задачи**:

— описать алгоритмы Левенштейна и Дамерау-Левенштейна для нахождения редакционного расстояния между строками;

— реализовать алгоритмы Левенштейна и Дамерау-Левенштейна;

— выполнить массированный замер времени работы алгоритмов;

— описать и обосновать полученные результаты в отчете.

1 Аналитическая часть

В данном разделе будут рассмотрены алгоритмы нахождения расстояний Левенштейна и Дамерау-Левенштейна.

1.1 Расстояние Левенштейна

Расстояние Левенштейна между двумя строками – это минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения строки в другую (каждая операция имеет свою цену – штраф) [1].

Пусть дано - строка $S1$, длина которой $L1$ и строка $S2$, длина которой $L2$.

Введем следующие обозначения:

— **I** (от английского слова Insert) – вставка символа в произвольной позиции ($w(\lambda, L2) = 1$);

— **D** (от английского слова Delete) – удаление символа в произвольной позиции ($w(L1, \lambda) = 1$);

— **R** (от английского слова Replace) – замена одного символа на другой символ ($w(L1, L2) = 1, L1 \neq L2$);

— **M** (от английского слова matrix) – совпадение двух символов ($w(L1, L2) = 0$).

1.2 Рекурсивный алгоритм нахождения расстояния Левенштейна

Пусть имеется две строки S_1 и S_2, длинной m и n соответственно. Расстояние Левенштейна $d(S_1, S_2) = D(m, n)$ рассчитывается по следующей рекуррентной формуле, с учетом введенных обозначений 1.1:

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

где сравнение символов строк S_1 и S_2 производится следующим образом :

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

1.3 Матричный алгоритм нахождения расстояния Левенштейна

Рекурсивный алгоритм вычисления расстояния Левенштейна может быть не эффективен при больших i и j, так как множество раз вычисляется промежуточное значение $D(i,j)$, что приводит к увеличению времени выполнения программы.

Для оптимизации можно использовать матрицу как хранилище промежуточных значений. Матрица имеет размеры:

$$(length(S_1)+1) * ((length(S_2)+1), \quad (1.3)$$

где length(S) – длина строки S.

В ячейке $[i,j]$ матрицы хранится значение $D(S_1[1..i], S_2[1..j])$. Первому элементу матрицы присвоено значение 0. Вся матрица заполняется в соответствии с соотношением 1.4:

$$A(i,j) = min \begin{cases} A(i-s1,j)+1 \\ A(i,j-1)+1 \\ A(i-1,j-1)+m(S_1[i],S_2[i]) \end{cases} \quad (1.4)$$

где где сравнение символов строк S_1 и S_2 производится следующим образом :

$$m(a,b) = \begin{cases} 0 & \text{если a = b,} \\ 1 & \text{иначе} \end{cases}. \quad (1.5)$$

Результат вычисления расстояния Левенштейна будет ячейка матрицы с индексами $i = length(S1)$ и $j = length(S2)$.

1.4 Расстояние Левенштейна с использованием кеша

Для оптимизации рекурсивного алгоритма заполнения можно использовать кеш в виде матрицы. Основная идея заключается в том, что при выполнении рекурсии происходит одновременное заполнение матрицы. Если рекурсивный алгоритм выполняет прогон для данных, которые еще не были обработаны, то результат нахождения заносится в матрицу. В случае, если обработанные данные встречаются снова, то для них расстояние не находится, и алгоритм переходит к следующему шагу.

1.5 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна между двумя строками(каждая из которых состоит из конечного числа символов) — это минимальное число операций вставки, удаления, замены одного символа и транспозиции двух соседних символов, необходимых для перевода одной строки в другую.

Данный алгоритм является модификацией алгоритма расстояния Ле-

венштейна, в него добавлена операции транспозиции (перестановки, двух символов).

Расстояние Дамерау-Левенштейна задается следующей рекуррентной формулой 1.6:

$$
D(m,n) = \\
= \begin{cases}
0, & \text{i} = 0, \text{j} = 0 \\
i, & \text{j} = 0, \text{i} > 0 \\
j, & \text{i} = 0, \text{j} > 0 \\
\min \begin{cases}
D(i,j-1)+1, \\
D(i-1,j)+1, \\
D(i-1,j-1), \\
D(i-2,j-2)+1,
\end{cases} & \text{если } i,j > 1, \\
& S_1[i] = S_2[j-1], \\
& S_1[i-1] = S_2[j], \\
\min \begin{cases}
D(i-1,j)+1, \\
D(i,j-1)+1, \\
D(i-1,j-1)+\text{m}(S_1[i], S_2[j])
\end{cases} & \text{иначе.}
\end{cases} \quad (1.6)
$$

Вывод

В данном разделе были теоретически описаны формулы Левенштейна и Дамерау-Левенштейна.

2 Конструкторская часть

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

2.1 Схемы алгоритмов

На вход алгоритмов подаются две строки s1 и s2, на выходе получаем искомое расстояние как целое число.

На рисунке 2.1 изображена рекурсивная реализация алгоритма поиска расстояния Левенштейна.

На рисунке 2.2 изображена схема матричного алгоритма поиска расстояния Левенштейна.

На рисунке 2.3 изображена рекурсивная реализация алгоритма поиска расстояния Левенштейна с использованием матрицы-кэша.

На рисунке 2.4 изображена схема матричной реализации алгоритма поиска расстояния Дамерау-Левенштейна.

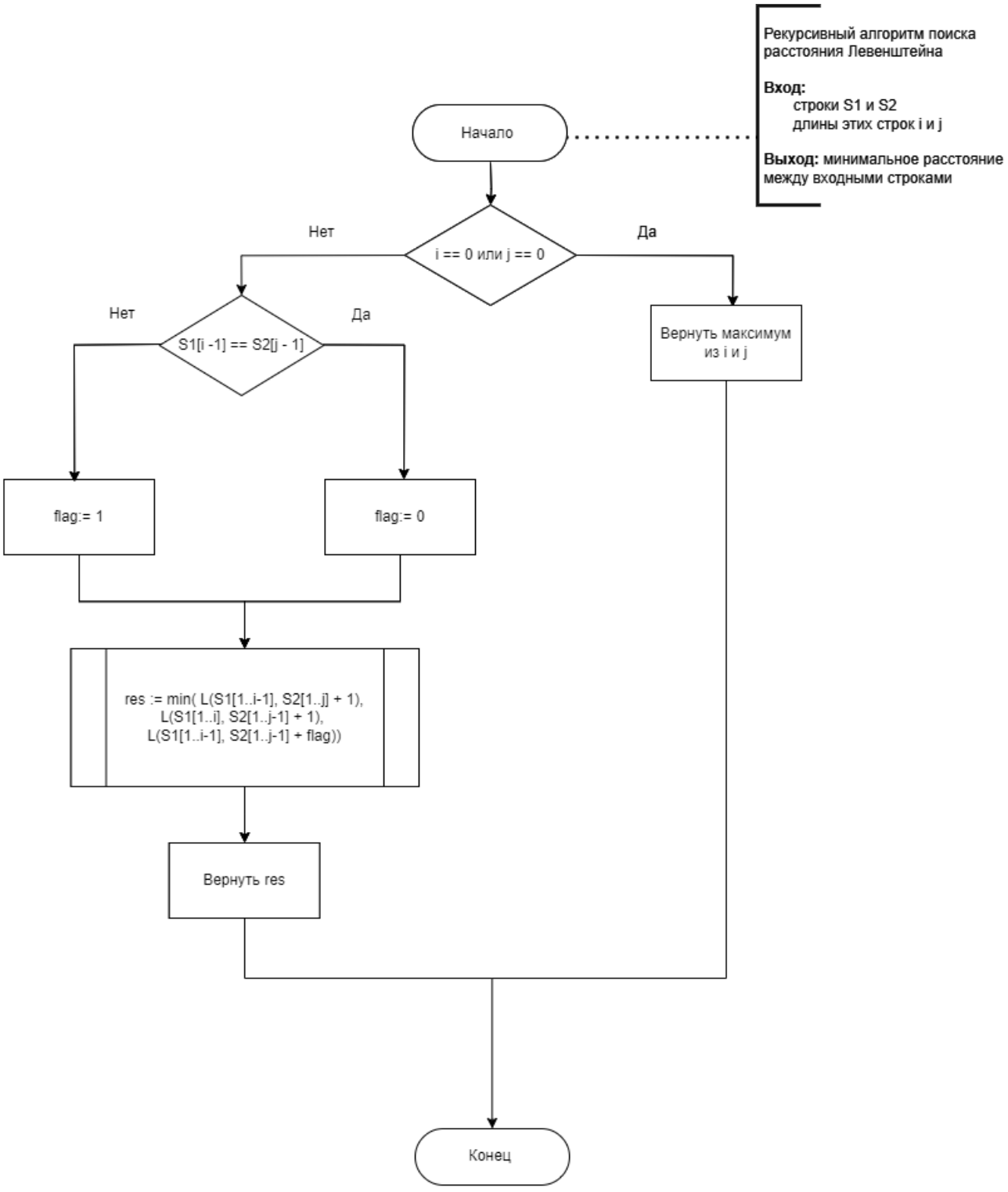

Рисунок 2.1 – Схема рекурсивного алгоритма Левенштейна

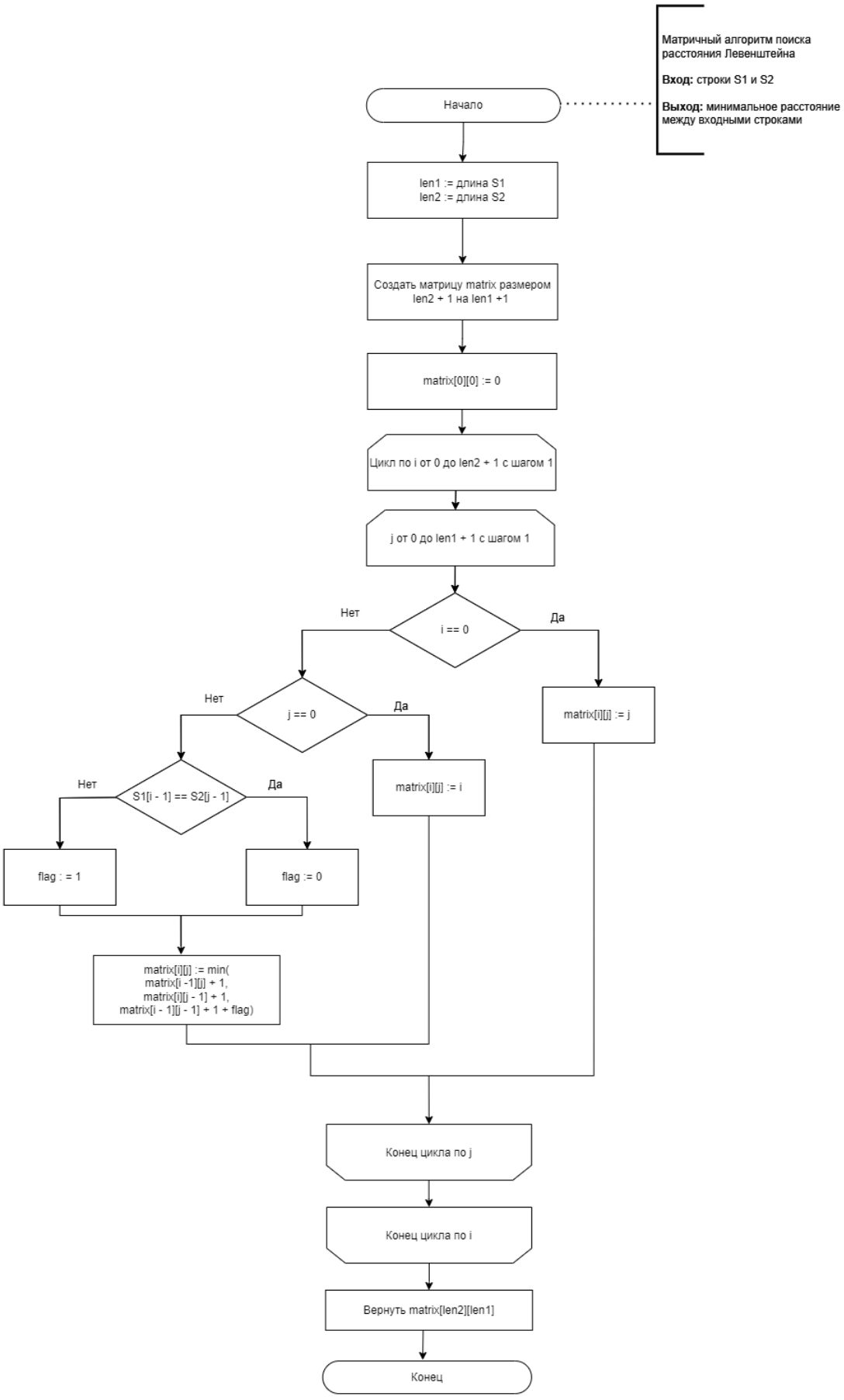

Рисунок 2.2 – Схема матричного алгоритма Левенштейна

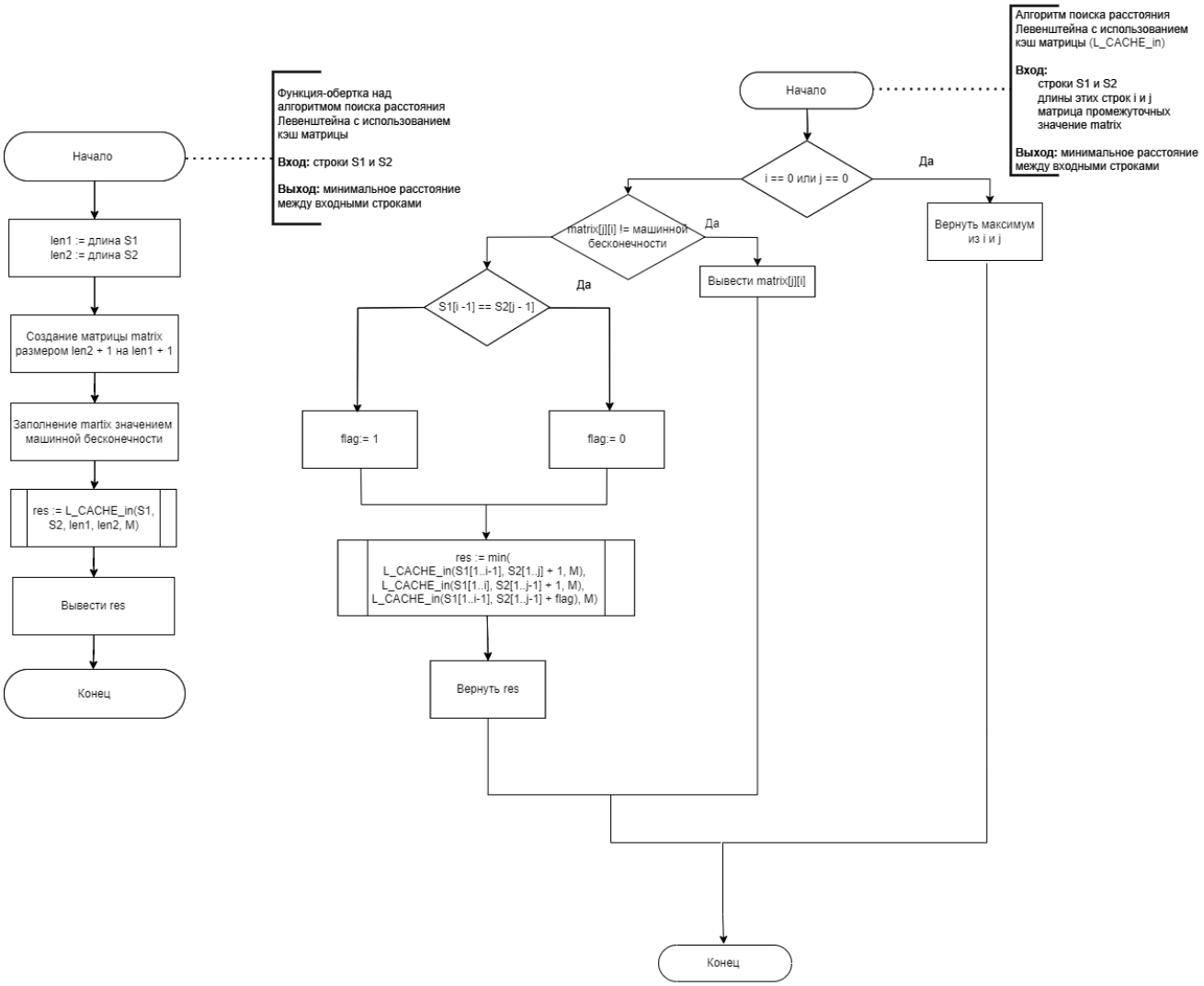

Рисунок 2.3 – Схема рекурсивного алгоритма Левенштейна с использованием кэша

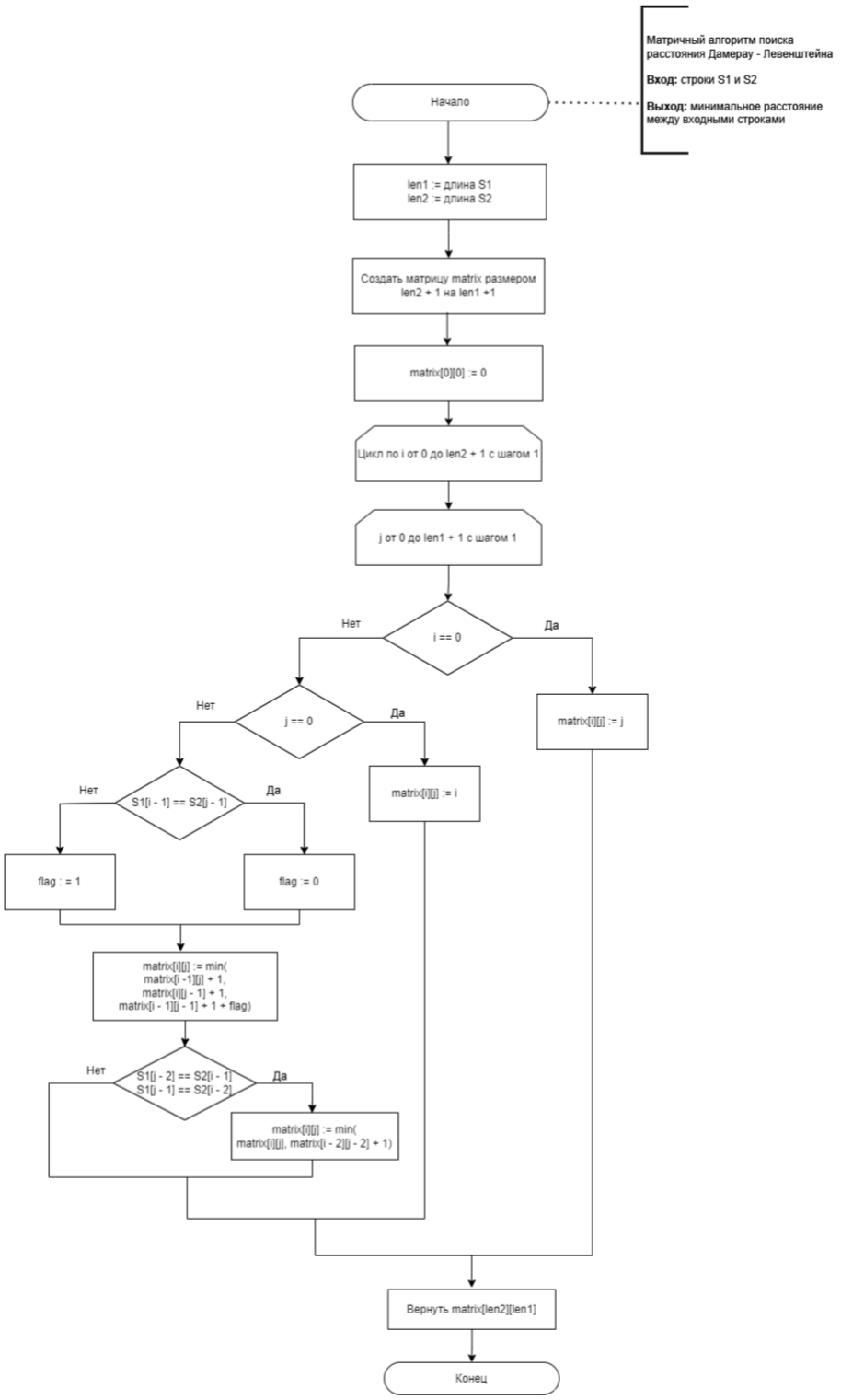

Рисунок 2.4 – Схема матричного алгоритма Дамерау-Левенштейна

2.2 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие структуры данных:

— строка — массив символов типа $wchar_t$;

— длина строки — целое число типа int.

Для матричной реализации алгоритма Левенштейна и рекурсивной реализации с кешем также будет использоваться матрица, которая является двумерным списком типа int.

2.3 Требования к программному обеспечению

К программе предъявлен ряд функциональных требований:

— наличие интерфейса для выбора действий из представленного в меню;

— возможность ввода строк;

— возможность обработки строк,написанных как на латинице так и на кириллице;

— возможность произвести замеры процессорного времени работы алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.

2.4 Требования к вводу

К входным данным предъявляются следующие требования:

1) На вход реализованным алгоритмам подаются две строки.

2) Строки могут быть пустыми.

3) Строки могут включать как латинские, так и кириллические символы.

4) Буквы нижнего и верхнего регистра считаются разными символами.

Вывод

В данном разделе на основе теоретических данных были определены требования к ПО. Также были построены схемы алгоритмов на основе данных, полученных на этапе анализа.

3 Технологическая часть

В данном разделе приведены средства реализации программного обеспечения, сведения о модулях программы, листинг кода и функциональные тесты.

3.1 Средства реализации

В качестве языка программирования, для написания данной лабораторной работы, был выбран C++, так как в нем имеется контейнер `std::wstring`, представляющий собой массив символов `std::wchar_t`, и библиотека `<ctime>` [4], позволяющая производить замеры процессорного времени.

3.2 Сведения о модулях программы

Данная программа разбита на следующие модули:

1) main.cpp — файл, содержащий точку входа в программу;
2) Matrix.cpp — файл, содержащий функции создания матрицы, ее освобождения и вывода на экран;
3) algorithms.cpp — файл, содержащий реализации алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна;
4) measure.cpp — файл, содержащий функции, замеряющие процессорное время выполнения алгоритмов.

3.3 Листинги кода

В листингах представлены реализации алгоритмов нахождения расстояния Левенштейна и Дамерау–Левенштейна.

Листинг 3.1 – Нахождения расстояния Левенштейна с помощью рекурсии

```
1  int levenstein_recursive(const std::wstring &word1, const
       std::wstring &word2, int len1, int len2)
2  {
3      if (len1 == 0 || len2 == 0)
4          return std::max(len1, len2);
5
6      int flag = (word1[len1 - 1] == word2[len2 - 1]) ? 0 : 1;
7
8      return min(
9          levenstein_recursive(word1, word2, len1, len2 - 1) + 1,
10         levenstein_recursive(word1, word2, len1 - 1, len2) + 1,
11         levenstein_recursive(word1, word2, len1 - 1, len2 - 1)
             + flag);
12 }
```

Листинг 3.2 – Матричный алгоритм поиска расстояния Левенштейна (часть 1)

```
1  int levenstein_not_recursive(const std::wstring &word1, const
       std::wstring &word2, bool out)
2  {
3      int len1 = word1.length();
4      int len2 = word2.length();
5      int **matrix = Matrix::Allocate(len2 + 1, len1 + 1,
           INT_MAX);
6      if (!matrix)
7          return -1;
8      matrix[0][0] = 0;
9      for (int i = 0; i <= len2; ++i)
10     {
11         for (int j = 0; j <= len1; ++j)
12         {
13             if (i == 0)
14             {
15                 matrix[i][j] = j;
16             }
17             else if (j == 0)
18             {
19                 matrix[i][j] = i;
20             }
21             else
22             {
23                 int flag = (word1[j - 1] == word2[i - 1]) ? 0 :
                       1;
24                 matrix[i][j] = min(
25                     matrix[i - 1][j] + 1,
26                     matrix[i][j - 1] + 1,
27                     matrix[i - 1][j - 1] + flag);
28             }
29         }
30     }
31     if (out)
32         Matrix::Print(matrix, word1, word2);
33     int res = matrix[len2][len1];
34     Matrix::Free(matrix, len2 + 1);
35
36     return res;
```

Листинг 3.3 – Вспомогательная функция нахождения расстояния Левенштейна с кешированием (часть 1)

```
1  static int recursive_for_levenstein_cache(const std::wstring
       &word1, const std::wstring &word2, int ind1, int ind2, int
       **matrix)
2  {
3      if (ind1 == 0)
4          return ind2;
5
6      if (ind2 == 0)
7          return ind1;
8
9      if (matrix[ind2][ind1] != INT_MAX)
10         return matrix[ind2][ind1];
11
12     int flag = (word1[j - 1] == word2[i - 1]) ? 0 : 1;
```

Листинг 3.4 – Вспомогательная функция нахождения расстояния Левенштейна с кешированием (часть 2)

```
1      int res = min(
2          recursive_for_levenstein_cache(word1, word2, ind1, ind2
             - 1, matrix) + 1,
3          recursive_for_levenstein_cache(word1, word2, ind1 - 1,
             ind2, matrix) + 1,
4          recursive_for_levenstein_cache(word1, word2, ind1 - 1,
             ind2 - 1, matrix) + flag
5      );
6
7      matrix[ind2][ind1] = res;
8      return res;
9  }
```

Листинг 3.5 – Нахождения расстояния Левенштейна с кешированием (оберточная функция)

```
1  int levenstein_cache_matrix(const std::wstring &word1, const
       std::wstring &word2)
2  {
3      int len1 = word1.length();
4      int len2 = word2.length();
5
6      int **matrix = Matrix::Allocate(len2 + 1, len1 + 1,
           INT_MAX);
7      if (!matrix)
8          return -1;
9
10     int res = recursive_for_levenstein_cache(word1, word2,
           len1, len2, matrix);
11
12     matrix::Free(matrix, len2 + 1);
13     return res;
14 }
```

Листинг 3.6 – Матричный алгоритм поиска расстояния Дамерау-Левенштейна (часть 1)

```
1  int damerau_levenstein_recursive(const std::wstring &word1,
       const std::wstring &word2, bool out)
2  {
3      int len1 = word1.length();
4      int len2 = word2.length();
5
6      int **matrix = Matrix::Allocate(len2 + 1, len1 + 1);
7      if (!matrix)
8          return -1;
9
10     matrix[0][0] = 0;
11     for (int i = 0; i <= len2; ++i)
12     {
```

Листинг 3.7 – Матричный алгоритм поиска расстояния Дамерау-Левенштейна (часть 2)

```
1         for (int j = 0; j <= len1; ++j)
2         {
3             if (i == 0)
4             {
5                 matrix[i][j] = j;
6             }
7             else if (j == 0)
8             {
9                 matrix[i][j] = i;
10            }
11            else
12            {
13                int flag = (word1[j - 1] == word2[i - 1]) ? 0 :
                      1;
14
15                matrix[i][j] = min(
16                    matrix[i - 1][j] + 1,
17                    matrix[i][j - 1] + 1,
18                    matrix[i - 1][j - 1] + flag);
19
20                if (word1[j - 2] == word2[i - 1] && word1[j -
                      1] == word2[i - 2])
21                {
22                    matrix[i][j] = std::min(
23                        matrix[i][j],
24                        matrix[i - 2][j - 2] + 1);
25                }
26            }
27        }
28    }
29
30    if (out)
31        Matrix::Print(matrix, word1, word2);
32
33    int res = matrix[len2][len1];
34    Matrix::Free(matrix, len2 + 1);
35
36    return res;
37 }
```

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	Ожидаемый результат	
		Левенштейн	Дамерау-Левенштейн
[]	[]	0	0
[]	подарок	7	7
кукла	кукла	0	0
скат	кот	2	2
heart	earth	2	2
вагон	гонки	4	4
рок	раб	2	2
птица	птицы	1	1

Вывод

В данном разделе были разработаны алгоритмы поиска расстояний Левенштейна (итеративно, рекурсивно, рекурсивно с кэшированием) и Дамерау-Левенштейна(итеративно) , а также проведено тестирование реализованных алгортимов.

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование, представлены далее.

— Архитектура: 32-битный микроконтроллер с тактовой частотой до 216 МГц;

— Память: Flash 2 Мбайт, RAM 512 Кбайт;

— LSE кварц: 32.768 кГц кварцевый генератор;.

При тестировании использовалась отладочная плата на базе MCU STM32F767ZIT6 (ARM Cortex-M7), ST-LINK/V2-1, Arduino, Ethernet.

4.2 Демонстрация работы программы

На рисунке 4.1 показан пример работы разработанной программы для случая, когда пользователь выбирает действие «Запуск алгоритмов поиска расстояния Левенштейна» и вводит строки «носорог» и «рогатка».

```
root@DESKTOP-G219U2V:~/aa_labs/lab1/lsi22r040/prog/build# ./app

                Меню
1. Запуск алгоритмов поиска расстояния Левенштейна:
   1) Рекурсивный Левенштейн.
   2) Нерекурсивный Левенштейна.
   3) Левенштейн с кэшем.
   4) Нерекурсивный Дамера-Левенштейн.
2. Замерить время для реализованных алгоритмов.
0. Выход

Выберите опцию (0-2): 1

Введите первое слово: носорог
Введите второе слово: рогатка

   1) Рекурсивный Левенштейн : 6
            н о с о р о г
          0 1 2 3 4 5 6 7
        р 1 1 2 3 4 4 5 6
        о 2 2 1 2 3 4 4 5
        г 3 3 2 2 3 4 5 4
        а 4 4 3 3 3 4 5 5
        т 5 5 4 4 4 4 5 6
        к 6 6 5 5 5 5 5 6
        а 7 7 6 6 6 6 6 6
   2) Нерекурсивный Левенштейна: 6
   3) Левенштейн с кэшем: 6
            н о с о р о г
          0 1 2 3 4 5 6 7
        р 1 1 2 3 4 4 5 6
        о 2 2 1 2 3 4 4 5
        г 3 3 2 2 3 4 5 4
        а 4 4 3 3 3 4 5 5
        т 5 5 4 4 4 4 5 6
        к 6 6 5 5 5 5 5 6
        а 7 7 6 6 6 6 6 6
   4) Нерекурсивный Дамерау-Левенштейна: 6
```

Рисунок 4.1 – Пример работы программы

4.3 Время выполнения алгоритмов

Замеры времени проводились на строках одинаковой длины.

Так как время работы алгоритмов может колебаться, в связи с различными процессами, происходящими в системе, для обеспечения более верных результатов измерения повторялись 500 раз, и бралось их среднее арифметическое значение.

На рисунке 4.2 показаны зависимости времени выполнения матричных реализаций алгоритмов Левенштейна и Дамерау-Левенштейна от длин входящих строк.

На рисунке 4.3 показаны зависимости времени выполнения рекурсивных реализаций алгоритмов Левенштейна и Дамерау-Левенштейна от длин входящих строк.

На рисунке 4.4 показаны зависимости времени выполнения рекурсивных реализаций алгоритма Дамерау-Левенштейна от длин входящих строк.

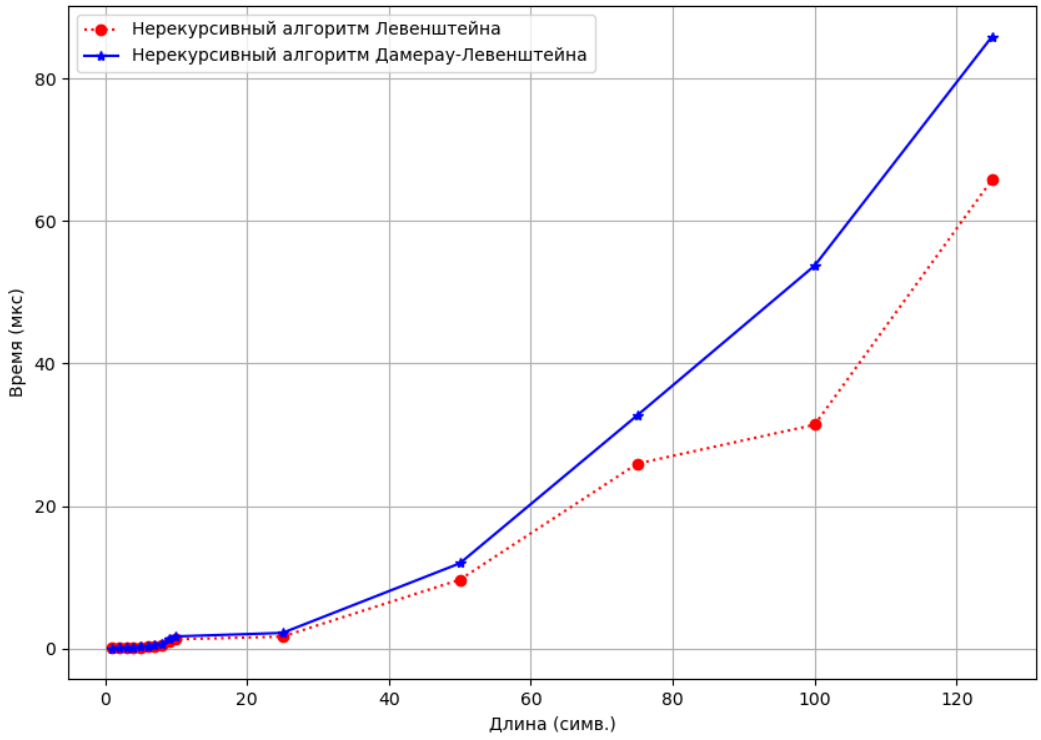

Рисунок 4.2 – Результат измерений времени работы нерекурсивных реализаций алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна

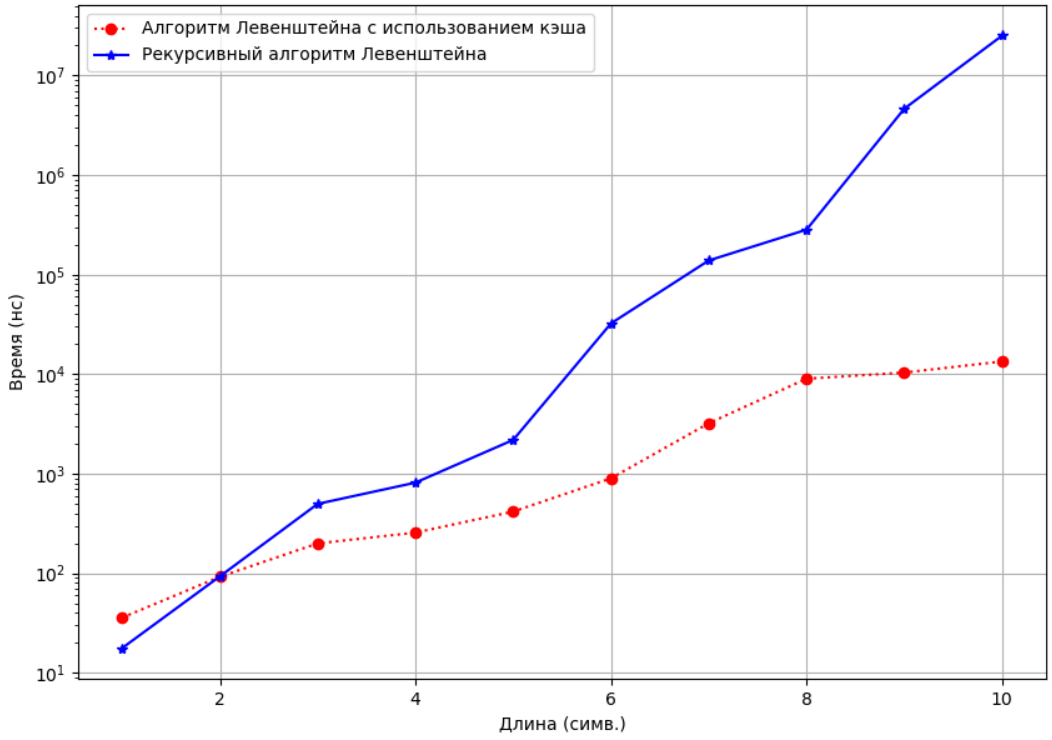

Рисунок 4.3 – Результат измерений времени работы рекурсивных реализаций алгоритма поиска расстояния Дамерау-Левенштейна

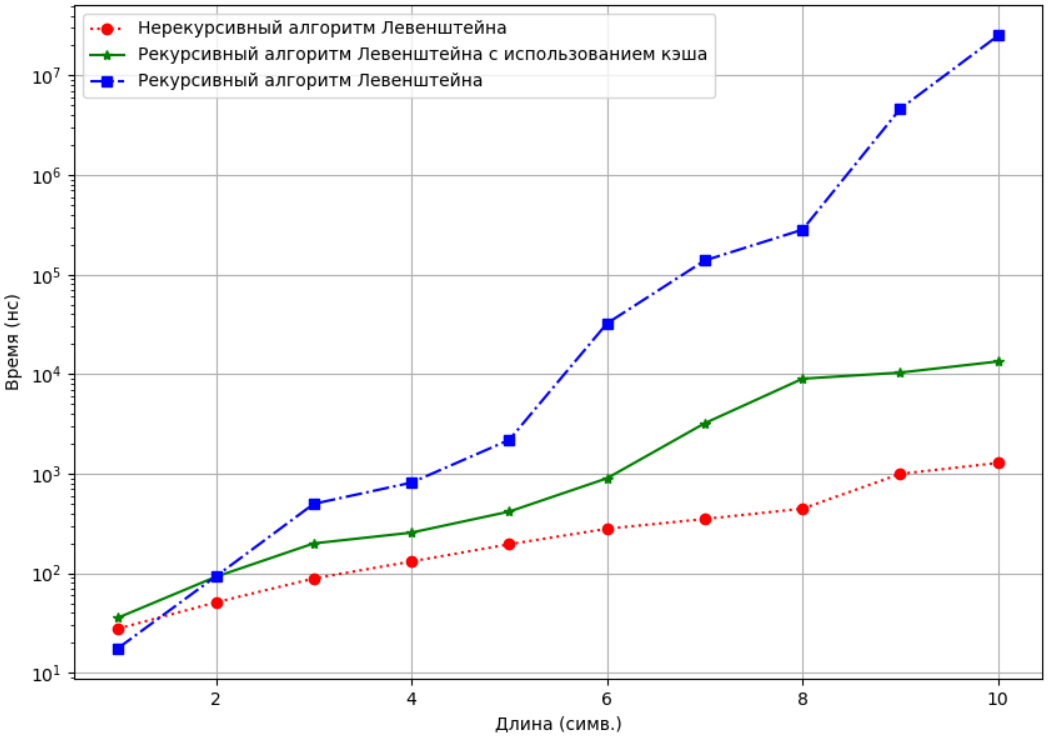

Рисунок 4.4 – Результат измерений времени работы реализаций алгоритмов поиска расстояния Дамерау-Левенштейна

Наиболее эффективными являются алгоритмы, использующие матрицы, так как в рекурсивных алгоритмах большое количество повторных расчетов.

Вывод

В результате исследования реализуемых алгоритмов по времени выполнения можно сделать следующие выводы:

— при больших длинах обрабатываемых строк алгоритм поиска расстояния Дамерау-Левенштейна выполняется на порядок дольше, что связано с обработкой дополнительного условия о перестановке символов;

— рекурсивная реализация алгоритма поиска расстояния Левенштейна с использованием кэша работает быстрее, реализации поиска этого расстояния без кэширования;

— время работы матричной и рекурсивной с кэшем реализаций алгоритма поиска расстояния Левенштейна приблизительно равны, и выполняются на порядок быстрее, в сравнении с рекурсивной реализацией поиска этого расстояния без кэширования.

ЗАКЛЮЧЕНИЕ

Экспериментально было подтверждено различие во временной эффективности рекурсивной и нерекурсивной реализаций алгоритмов, нахождения расстояния между строками, на основе разработаного программного обеспечения и замеров процессорного времени выполнения реализаций на различных длинах строк.

Из полученных данных можно сделать вывод о том, что матричная реализация данных алгоритмов заметно выигрывает по времени при росте длин строк.

В ходе выполнения данной лабораторной работы были решены следующие задачи:

— описаны алгоритмы Левенштейна и Дамерау-Левенштейна для нахождения расстояния между строками;

— реализованы алгоритмы Левенштейна и Дамерау-Левенштейна;

— проведен сравнительный анализ линейной и рекурсивной реализаций алгоритмов определения расстояния между строками по затрачиваемым ресурсам времени;

— экспериментально подтверждено различие во временной эффективности рекурсивной и нерекурсивной реализаций алгоритмов:

  – при больших длинах обрабатываемых строк алгоритм поиска расстояния Дамерау-Левенштейна выполняется на порядок дольше, что связано с обработкой дополнительного условия о перестановке символов;

  – рекурсивная реализация алгоритма поиска расстояния Левенштейна с использованием кэша работает быстрее, реализации поиска этого расстояния без кэширования;

  – время работы матричной и рекурсивной с кэшем реализаций алгоритма поиска расстояния Левенштейна приблизительно равны, и выполняются на порядок быстрее, в сравнении с рекурсивной реализацией поиска этого расстояния без кэширования.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. А. Погорелов Д., М. Таразанов А. Сравнительный анализ алгоритмов редакционного расстояния Левенштейна и Дамерау-Левенштейна // Синергия Наук. 2019. https://elibrary.ru/item.aspid=36907767 (дата обращения 17.09.2024)

2. Документация по Microsoft C++ [Электронный ресурс]. Режим доступа: https://learn.microsoft.com/ru-ru/cpp/?view=msvc-170&viewFallbackFrom=vs-2017 (дата обращения: 17.09.2024).

3. Standard library header <ctime> [Электронный ресурс]. Режим доступа: https://en.cppreference.com/w/cpp/header/ctime (дата обращения: 17.09.2024).

4. Intel [Электронный ресурс]. Режим доступа: https://www.intel.ru/content/www/ru/ru/products/details/processors/core/i5.html (дата обращения: 17.09.2024).