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 Расстояние Левенштейна и Расстояние Дамерау Левенштейна

Студент Кузин А. А.

Группа ИУ7-51 Б

Преподаватель Волкова Л. Л.

Москва — 2023 г.

Содержание

Введение

Расстояние Левенштейна, или редакционное расстояние, — метрика сходства между двумя строковыми последовательностями. Чем больше расстояние, тем более различны строки. Для двух одинаковых последовательностей расстояние равно нулю. По сути, это минимальное число односимвольных преобразований (удаления, вставки или замены), необходимых, чтобы превратить одну последовательность в другую.

Расстояние Дамерау — Левенштейна является модификацией расстояния 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. К операциям вставки, удаления и замены символов, определённых в расстоянии Левенштейна добавлена операция транспозиции (перестановки) символов.

Расстояние Левенштейна активно используется для исправления ошибок в словах, поиска дубликатов текстов, сравнения геномов и прочих полезных операций с символьными последовательностями.

Целью данной лабораторной работы является получить навыки динамического программирования на задаче поиска редакционных расстояний.

Необходимо выполнить следующие **задачи**:

1. Изучить алгоритмы поиска расстояния Левенштейна и Дамерау — Левенштейна.

2. Разработать и реализовать алгоритмы поиска редакционных расстояний.

3. Выполнить замеры затрат реализаций алгоритмов по памяти.

4. Выполнить замеры затрат реализаций алгоритмов по процессорному времени.

5. Провести сравнительных анализ двух нерекурсивных алгоритмов.

1 Аналитическая часть

1.1 Расстояние Левенштейна

Расстояние Левенштейна — метрика, измеряющая по модулю разность между двумя последовательностями символов. Она определяется как минимальное количество односимвольных операций (а именно вставки, удаления, замены), необходимых для превращения одной последовательности символов в другую. В общем случае, операциям, используемым в этом преобразовании, можно назначить разные цены [1].

1. **I** (от англ. insert) — вставка символа ($w(\lambda, b) = 1$);
2. **R** (от англ. replace) — замена символа ($w(a, b) = 1$, $a \neq b$);
3. **D** (от англ. delete) — удаление символа ($w(a, \lambda) = 1$).
4. **M** (от англ. match) — совпадение символа ($w(a, a) = 0$);

Также рассмотрим функцию $D(i, j)$: ее значением является редакционное расстояние между строками $S_1[1...i]$ и $S_2[1...j]$.

Расстояние Левенштейна между двумя строками S_1 и S_2 (длиной M и N соответственно) рассчитывается по следующей рекуррентной формуле:

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

где сравнение символов строк S_1 и S_2 рассчитывается таким образом:

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

1.2 Расстояние Дамерау — Левенштейна

Расстояние Дамерау — Левенштейна между двумя строками, состоящими 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.

Расстояние Дамерау — Левенштейна определятся следующей рекуррентной формуле:

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

Вывод

В данном разделе были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау — Левенштейна.

2 Конструкторская часть

В данном разделе будут приведены схемы алгоритмов нахождения расстояний Левенштейна и Дамерау — Левенштейна, описание используемых типов данных и структуры программного обеспечения.

2.1 Требования к программному обеспечению

К программе предъявлен ряд функциональных требований:

- наличие интерфейса для выбора действий;
- возможность ввода строк с клавиатуры;
- возможность обработки строк, состоящих из символов латинского алфавита или арабских цифр;
- возможность произвести замеры процессорного времени работы реализованных алгоритмов поиска расстояний Левенштейна и Дамерау — Левенштейна;

2.2 Требования к вводу

К допустимому вводу представлены следующие ограничения:

1. На вход реализованным алгоритмам подаются две строки;
2. Строки могут включать символы латинского алфавита или арабских цифр;
3. Буквы нижнего и верхнего регистра считаются разными символами;

2.3 Разработка алгоритмов

На рисунке 2.1 представлена схема матричного алгоритма поиска расстояния Левенштейна.

На рисунке 2.2 приведена схема матричной реализации алгоритма поиска расстояния Дамерау — Левенштейна.

На рисунке 2.3 представлена его рекурсивная реализация.

На рисунке 2.5 показана рекурсивная реализация алгоритма нахождения расстояния Дамерау — Левенштейна с использованием матрицы-кэша.

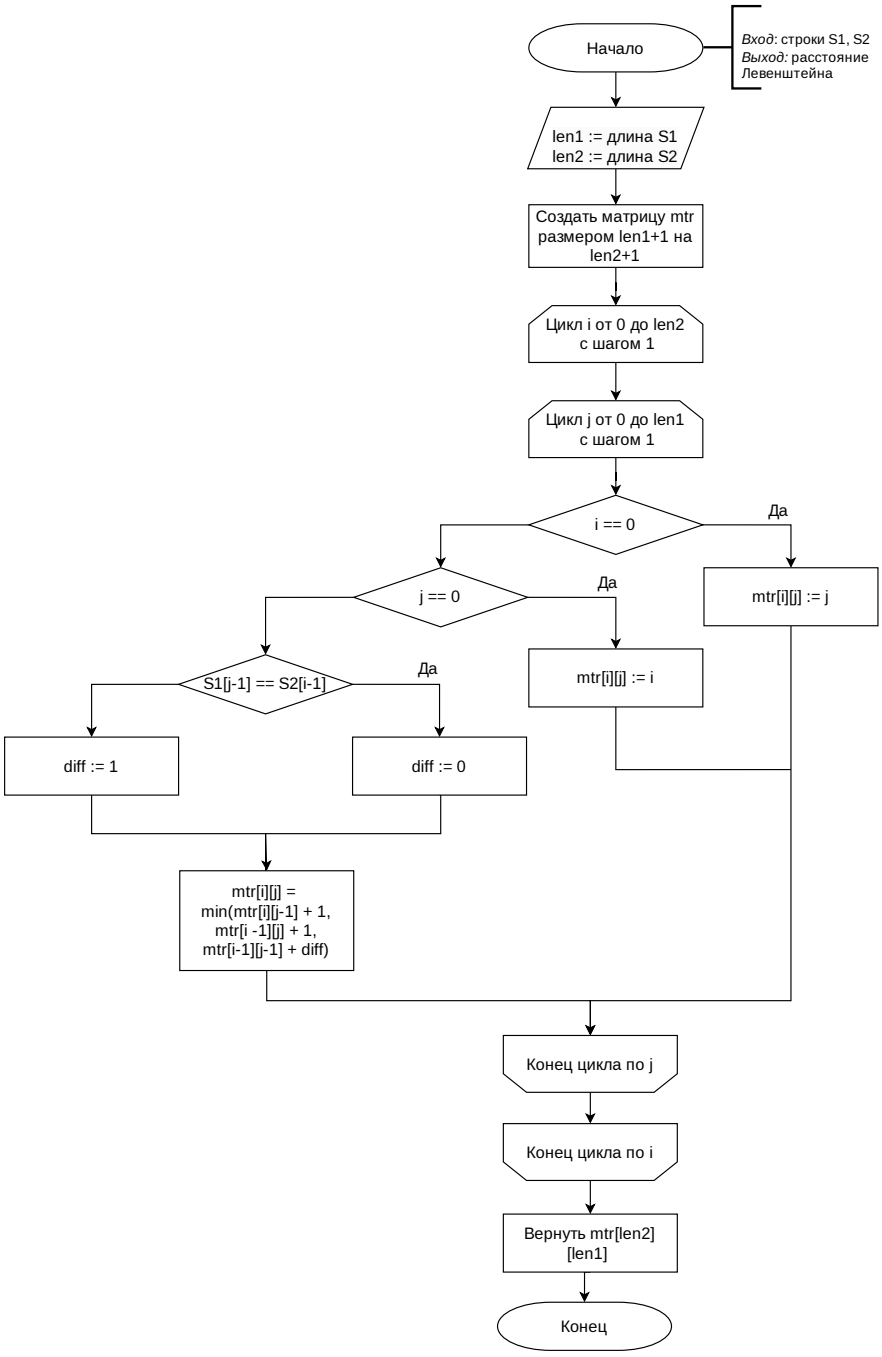

Рисунок 2.1 – Схема нерекурсивного алгоритма нахождения расстояния Левенштейна

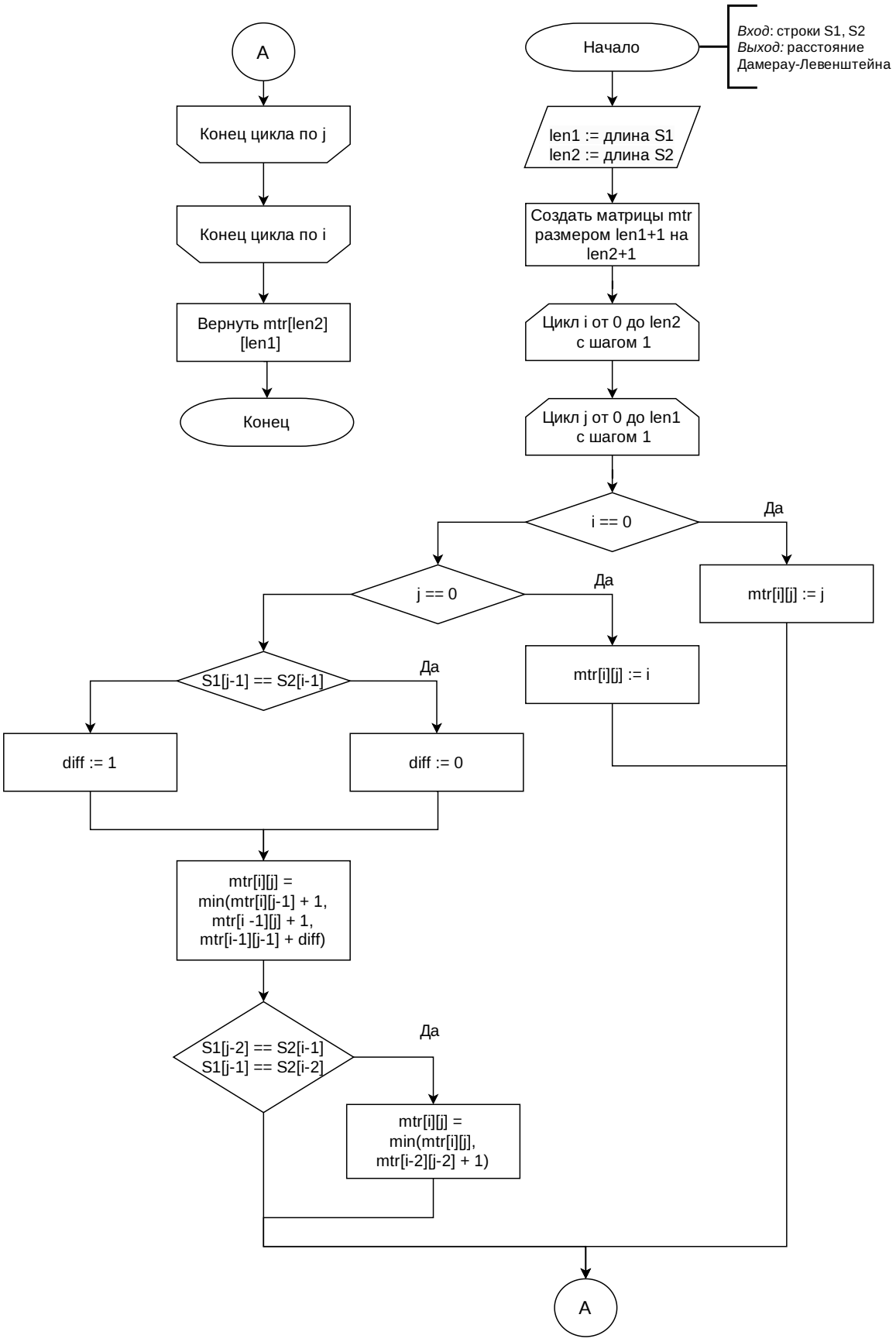

Рисунок 2.2 – Схема нерекурсивного алгоритма нахождения расстояния Дамерау — Левенштейна

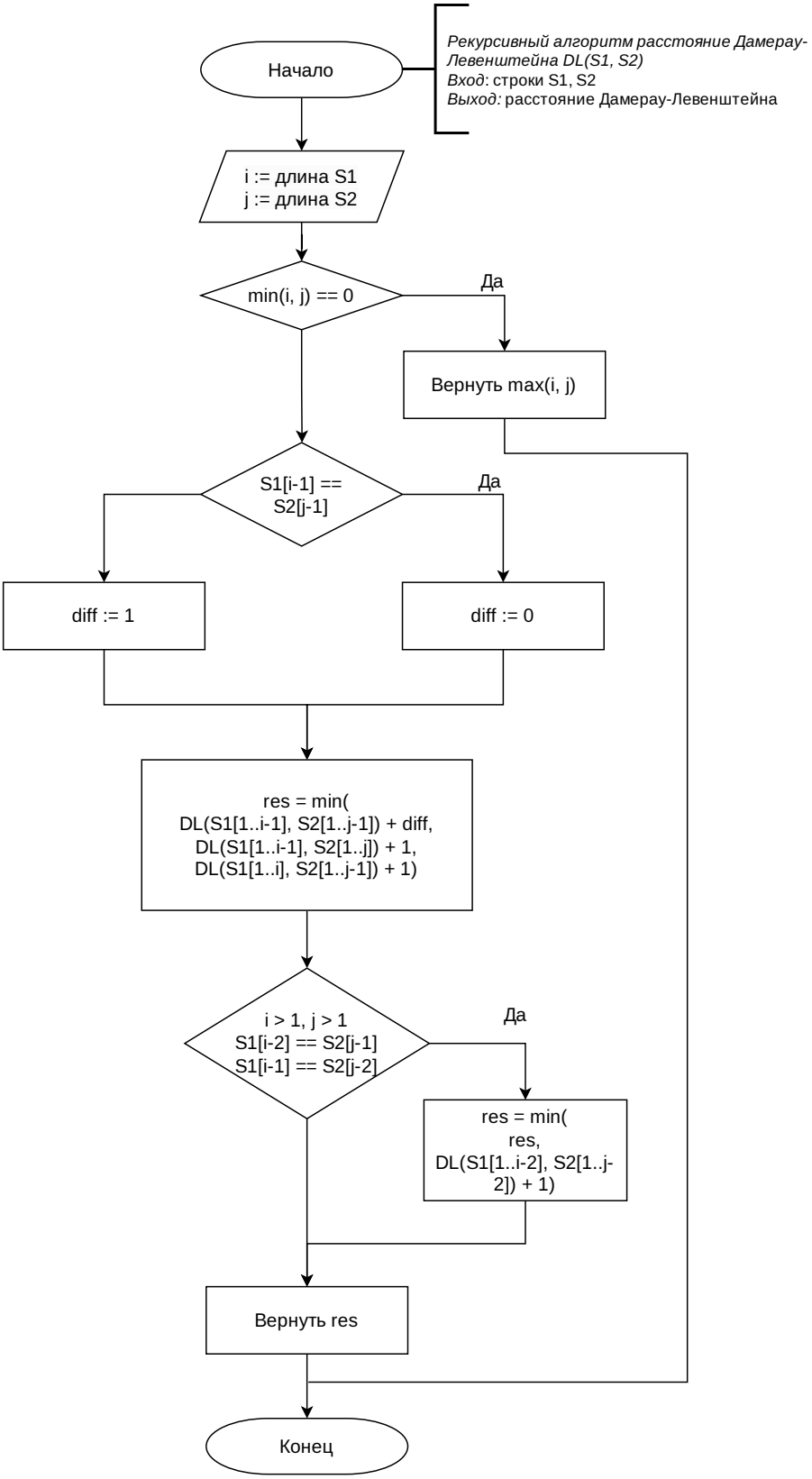

Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Дамерау — Левенштейна

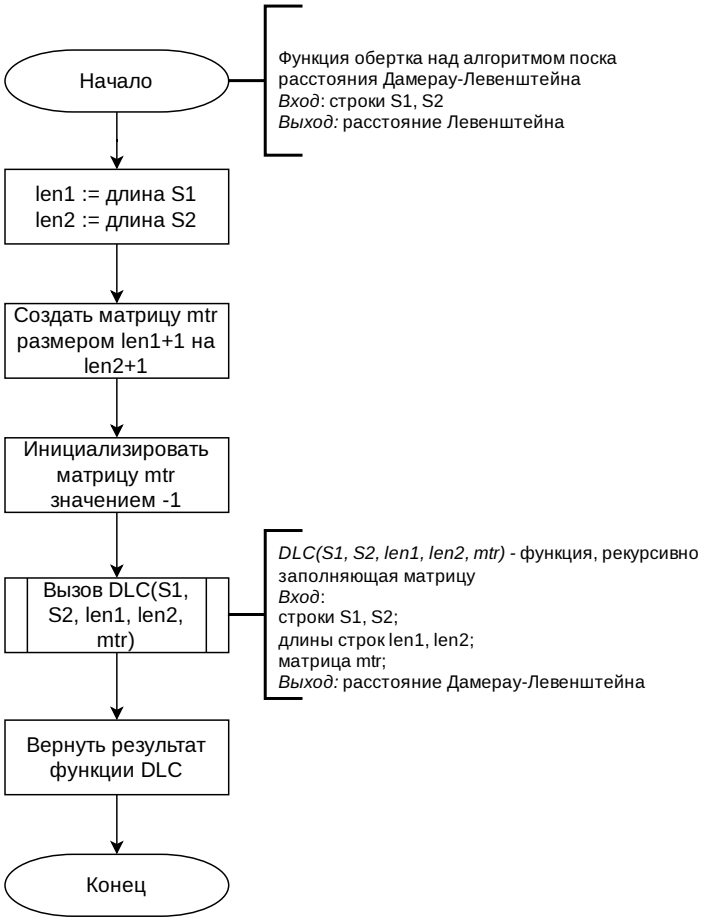

Рисунок 2.4 – Схема алгоритма вызова рекурсивного алгоритма нахождения расстояния Дамерау — Левенштейна с кешированием

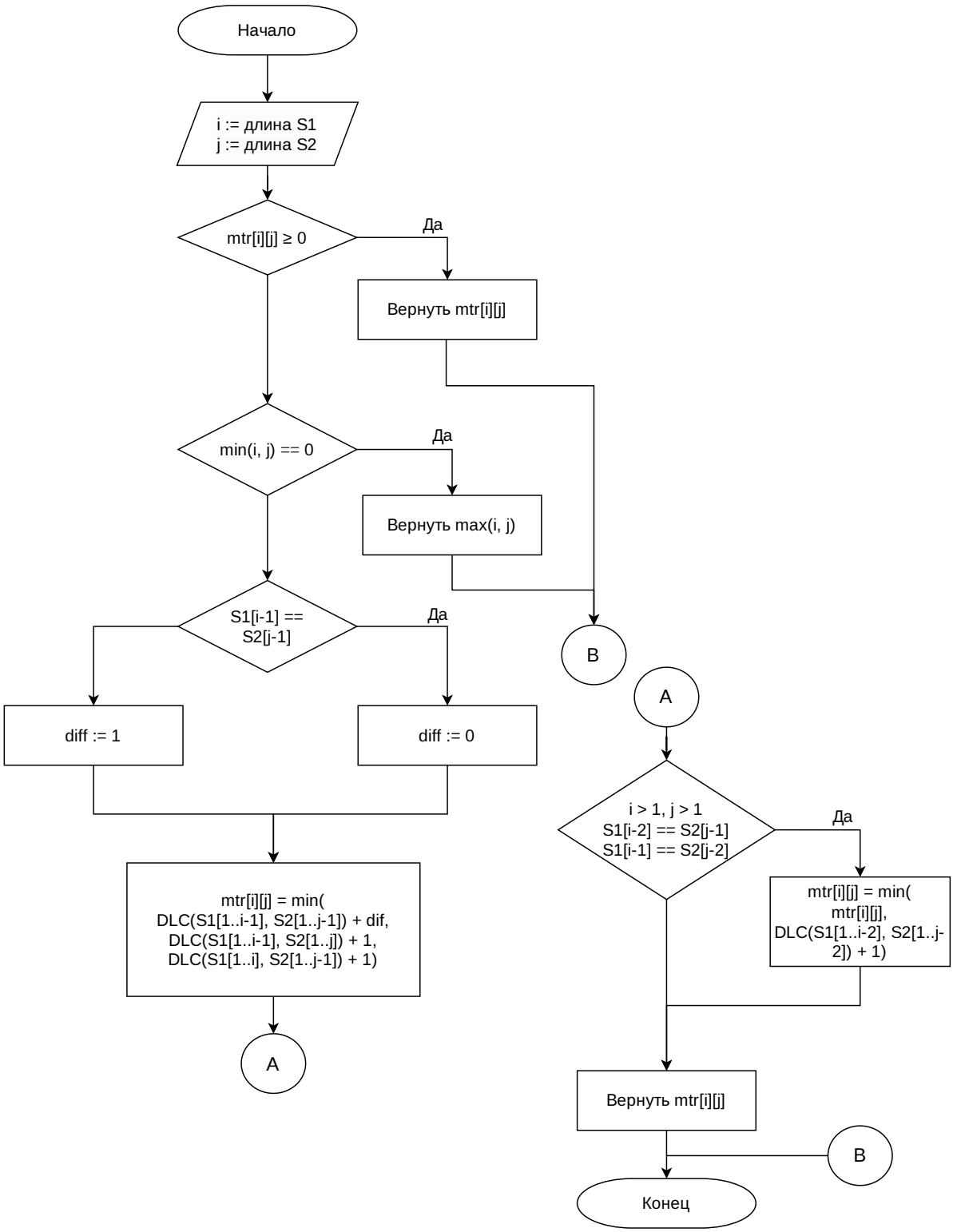

Рисунок 2.5 – Схема рекурсивного алгоритма нахождения расстояния Дамерау — Левенштейна с кешированием

2.4 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие типы данных:

- *строка* — массив `символьного типа`;
- *матрица* — двумерный массив `целочисленного типа`.

Вывод

В данном разделе на основе теоретических данных были перечислены требования к ПО и построены схемы реализуемых алгоритмов на основе данных, полученных на этапе анализа.

3 Технологическая часть

В данном разделе будут приведены средства реализации, листинг кода и функциональные тесты.

3.1 Средства реализации

В качестве языка программирования, используемого при написании данной лабораторной работы, был выбран C++ [2], так как в нем имеется контейнер `std::string`, представляющий собой массив символов `char`, и библиотека `<ctime>` [3], позволяющая производить замеры процессорного времени.

В качестве среды для написания кода был выбран *Visual Studio Code* за счет того, что она предоставляет функционал для проектирования, разработки и отладки ПО.

3.2 Сведения о модулях программы

Данная программа разбита на следующие модули:

- `main.cpp` — точка входа программы, пользовательское меню;
- `algo` — модуль с реализациями алгоритмов поисков расстояний;
- `measurement` — модуль с реализаций функции подсчёта затрачиваемого времени;

3.3 Реализация алгоритмов

Далее будет представлена реализация алгоритмов поиска редакционных растояний:

Листинг 3.1 – Матричный алгоритм поиска расстояния Левенштейна

```
1  int matrix_lev_temp(const char* str1,const char* str2, int width,
       int height)
2  {
3      int diff;
4
5      int** matrix = new int*[height];
6      for (int i = 0; i < height; ++i)
7          matrix[i] = new int[width];
8
9      for(int i = 0; i < width; i ++)
10     {
11         matrix[0][i] = i;
12     }
13     for(int i = 1; i < height; i ++)
14     {
15         matrix[i][0] = i;
16     }
17
18     for(int i = 1; i < height; i ++)
19     {
20         for(int j = 1; j < width; j ++)
21         {
22             int minval = min(matrix[i][j - 1], matrix[i - 1][j]);
23             minval ++;
24             minval = min(minval, matrix[i - 1][j - 1] +
                   int(str1[j] != str2[i]));
25
26             matrix[i][j] = minval;
27         }
28     }
29
30     if(MODE == MODE_INTERACTIVE)
31         print_matrix(matrix, width, height);
32
33     diff = matrix[height - 1][width - 1];
```

```
34
35      for (int i = 0; i < height; ++i)
36          delete[] matrix[i];
37      delete[] matrix;
38
39      return diff;
40  }
```

Листинг 3.2 – Матричный алгоритм поиска расстояния Дамерау — Левенштейна

```
1  int matrix_dam_lev_temp(const char* str1,const char* str2, int
       width, int height)
2  {
3      int diff;
4
5      int** matrix = new int*[height];
6      for (int i = 0; i < height; ++i)
7          matrix[i] = new int[width];
8
9
10     for(int i = 0; i < height; i ++)
11     {
12         for(int j = 0; j < width; j ++)
13         {
14             matrix[i][j] = 0;
15         }
16     }
17
18     for(int i = 0; i < width; i ++)
19     {
20         matrix[0][i] = i;
21     }
22     for(int i = 1; i < height; i ++)
23     {
24         matrix[i][0] = i;
25     }
26
27     for(int i = 1; i < width; i ++)
28     {
29             int minval = min(matrix[i][0], matrix[i - 1][1]);
30             minval ++;
31             minval = min(minval, matrix[i - 1][0] + int(str1[1]
                   != str2[i]));
32             matrix[i][1] = minval;
33     }
34
35     for(int j = 1; j < height; j ++)
36     {
37             int minval = min(matrix[1][j - 1], matrix[0][j]);
```

```
38              minval ++;
39              minval = min(minval, matrix[0][j - 1] + int(str1[j]
                    != str2[1]));
40              matrix[1][j] = minval;
41      }
42
43      for(int i = 2; i < height; i ++)
44      {
45          for(int j = 2; j < width; j ++)
46          {
47              int minval = min(matrix[i][j - 1], matrix[i - 1][j]);
48              minval ++;
49              minval = min(minval, matrix[i - 1][j - 1] +
                    int(str1[j] != str2[i]));
50              if(str1[j] == str2[i - 1] && str1[j - 1] == str2[i])
51              {
52                  minval = min(minval, matrix[i - 2][j - 2] + 1);
53              }
54              matrix[i][j] = minval;
55          }
56      }
57
58      if(MODE == MODE_INTERACTIVE)
59          print_matrix(matrix, width, height);
60
61      diff = matrix[height - 1][width - 1];
62
63      for (int i = 0; i < height; ++i)
64          delete[] matrix[i];
65      delete[] matrix;
66
67      return diff;
68  }
```

Листинг 3.3 – Рекурсивный алгоритм поиска расстояния Дамерау — Левенштейна

```
int recurs_dam_lev_temp(const char*str1, const char*str2, int
    length1, int length2)
{
    if (length1 == 0)
        return length2;
    if (length2 == 0)
        return length1;

    int change = 0;
    int res = 0;
    if (str1[length1 - 1] != str2[length2 - 1])
        change = 1;

    res = min(recurs_dam_lev_temp(str1, str2, length1, length2 -
        1) + 1,
                min(recurs_dam_lev_temp(str1, str2, length1 - 1,
                    length2) + 1,
                    recurs_dam_lev_temp(str1, str2, length1 - 1,
                        length2 - 1) + change));

    if (length1 > 1 && length2 > 1 &&
        str1[length1 - 1] == str2[length2 - 2] &&
        str1[length1 - 2] == str2[length2 - 1])
        res = min(res, recurs_dam_lev_temp(str1, str2, length1 -
            2, length2 - 2) + 1);
    return res;
}
```

Листинг 3.4 – Рекурсивный алгоритм поиска расстояния Дамерау — Левенштейна с кэшированием (реализация)

```
int hash_dam_lev_temp(const char*str1, const char*str2, int
   **matrix, int width, int height)
{
    if (height == 0)
        return matrix[height][width] = width;
    if (width == 0)
        return matrix[height][width] = height;

    int change = 0;
    if (str1[height - 1] != str2[width - 1])
        change = 1;


    matrix[height][width] = min(hash_dam_lev_temp(str1, str2,
       matrix, height, width - 1) + 1,
                  min(hash_dam_lev_temp(str1, str2, matrix,
                     height - 1, width) + 1,
                      hash_dam_lev_temp(str1, str2, matrix,
                         height - 1, width - 1) + change));

    if (height > 1 && width > 1 &&
        str1[height - 1] == str2[width - 2] &&
        str1[height - 2] == str2[width - 1])
        matrix[height][width] = min(matrix[height][width],
           hash_dam_lev_temp(str1, str2, matrix, height - 2,
           width - 2) + 1);

    return matrix[height][width];
}
```

Листинг 3.5 – Рекурсивный алгоритм поиска расстояния Дамерау — Левенштейна с кэшированием (оберточная функция)

```
1  template <IsStr T>
2  int hash_dam_lev(const T& str1,const T& str2)
3  {
4
5      size_t width = str1.length() + 1;
6      size_t height = str2.length() + 1;
7
8      int** matrix = new int*[height];
9      for (int i = 0; i < height; i++)
10         matrix[i] = new int[width];
11
12     for (int i = 0; i < height; i++)
13     {
14         for (int j = 0; j < width; j++)
15         {
16                 matrix[i][j] = 0;
17         }
18     }
19     int diff = hash_dam_lev_temp(str1.c_str(), str2.c_str(),
           matrix, width - 1, height - 1);
20
21     for (int i = 0; i < height; ++i)
22         delete[] matrix[i];
23     delete[] matrix;
24
25     return diff;
26
27 }
```

3.4 Функциональные тесты

В данном разделе будут представлены функциональные тесты, проверяющие работу алгоритмов поиска расстояний.

Таблица 3.1 – Функциональные тесты

Входные данные		**Расстояние и алгоритм**			
Строка 1	**Строка 2**	**Левенштейна**	**Дамерау — Левенштейна**		
		Итеративный	**Итеративный**	**Рекурсивный**	
				Без кеша	**С кешем**
kitten	sitting	4	4	4	4
book	back	2	2	2	2
back	bakc	2	1	1	1
saturday	sunday	3	3	3	3
intention	inteniton	2	1	1	1

Вывод

Были реализованы алгоритмы Левенштейна (итеративно) и Дамерау — Левенштейна (итеративно, рекурсивно, рекурсивно с кэшированием). Проведено тестирование реализованных алгортимов.

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры по времени:

- Процессор: Intel Core i7 9750H 2.6 ГГц.
- Оперативная память: 16 ГБ.
- Операционная система: Kubuntu 22.04.3 LTS x86_64 Kernel: 6.2.0-36-generic.

Во время проведения измерений времени ноутбук был подключен к сети электропитания и был нагружен только системными приложениями.

4.2 Демонстрация работы программы

На рисунке 4.1 показан пример работы с программой.

```
po4ti@Po4ti-PC:~/Рабочий стол/Lab_01_AA$ ./app.exe
1 - Левенштейн с использованием матрицы
2 - Дамерау - Левенштейн с использованием матрицы
3 - Рекурсивный алгоритм расстояния Дамерау - Левенштейн
4 - Дамерау - Левенштейн с использованием хэш таблицы
0 - Выход
3
Введите первую строку:
string
Введите вторую строку:
strng
Расстояние Дамерау - Левенштейна: 1
1 - Левенштейн с использованием матрицы
2 - Дамерау - Левенштейн с использованием матрицы
3 - Рекурсивный алгоритм расстояния Дамерау - Левенштейн
4 - Дамерау - Левенштейн с использованием хэш таблицы
0 - Выход
3
Введите первую строку:
str1ng1
Введите вторую строку:
string2
Расстояние Дамерау - Левенштейна: 2
1 - Левенштейн с использованием матрицы
2 - Дамерау - Левенштейн с использованием матрицы
3 - Рекурсивный алгоритм расстояния Дамерау - Левенштейн
4 - Дамерау - Левенштейн с использованием хэш таблицы
0 - Выход
```

Рисунок 4.1 – Демонстрация работы программы.

4.3 Временные характеристики

В связи с экспоненциальным ростом затрачиваемого времени рекурсивными алгоритмами, верхняя граница длин строк для тестирования таких алгоритмов ограничена до 10.

Итого, исследование временных характеристик реализованных алгоритмов производилось на строках длинами:

- 1 – 10 с шагом 1 для рекурсивных реализаций;
- 1 – 200 с шагом 1 для нерекурсивных реализаций.

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 строкк.

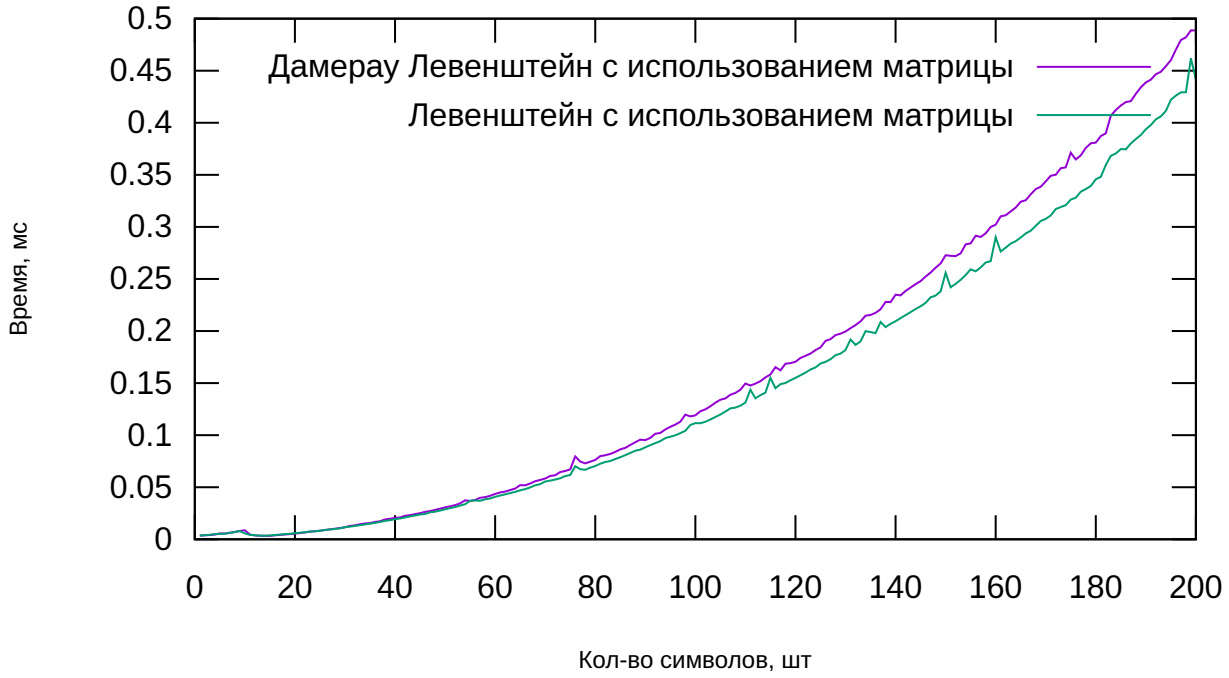

Рисунок 4.2 – Результат измерений времени работы (в мс) нерекурсивных реализаций алгоритмов поиска расстояний Левенштейна и Дамерау — Левенштейна

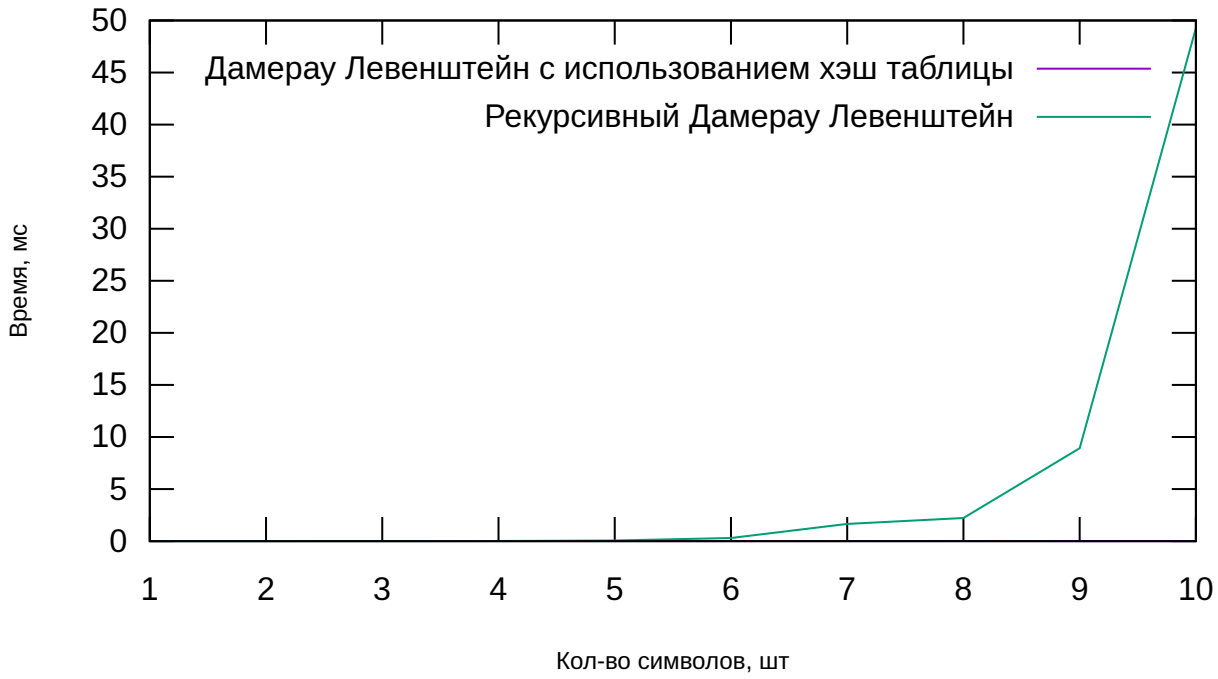

Рисунок 4.3 – Результат измерений времени работы рекурсивных реализаций алгоритма поиска расстояния Дамерау — Левенштейна

4.4 Характеристики по памяти

Введем следующие обозначения:

- m — длина строки S_1;
- n — длина строки S_2;
- size(v) — функция, вычисляющая размер входного параметра v в байтах;
- *char* — тип данных, используемый для хранения символа строки;
- *int* — целочисленный тип данных.

Теоретически оценим объем используемой памяти итеративной реализацией алгоритма поиска расстояния Левенштейна:

$$M_{LevIter} = (m+1)\cdot(n+1)\cdot \text{size}(int) + (m+n)\cdot \text{size}(char) + \\ + \text{size}(int**) + (m+1)\cdot \text{size}(int*) + \\ + 3\cdot \text{size}(int) + 2\cdot \text{size}(int) \quad (4.1)$$

где $(m+1)\cdot(n+1)\cdot \text{size}(int)$ — размер матрицы,
$\text{size}(int**)$ — размер указателя на матрицу,
$(m+1)\cdot \text{size}(int*)$ — размер указателей на строки матрицы,
$(m+n)\cdot \text{size}(char)$ — размер двух входных строк,
$2\cdot \text{size}(int)$ — размер переменных, хранящих длину строк,
$3\cdot \text{size}(int)$ — размер дополнительных переменных.

Для алгоритма поиска расстояния Дамерау — Левенштейна теоретическая оценка объема используемой памяти идентична.

Произведем оценку затрат по памяти для рекурсивных реализаций алгоритма нахождения расстояния Дамерау — Левенштейна.

Рассчитаем объем памяти, используемой каждым вызовом функции поиска расстояния Дамерау — Левенштейна:

$$M_{call} = (m+n)\cdot \text{size}(char) + 2\cdot \text{size}(int) + 3\cdot \text{size}(int) + 8 \quad (4.2)$$

где $(m+n)\cdot \text{size}(char)$ — объем памяти, используемый для хранения двух строк,
$2\cdot \text{size}(int)$ — размер двух входных строк,
$3\cdot \text{size}(int)$ — размер дополнительных переменных,
8 байт — адрес возврата.

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

$$M_{DLRec} = (m+n)\cdot M_{call} \quad (4.3)$$

где $m+n$ — максимальная глубина стека вызовов,
M_{call} — затраты по памяти для одного рекурсивного вызова.

Рекурсивная реализация алгоритма поиска расстояния Дамерау — Левенштейна с кэшированием для хранения промежуточных значений исполь-

зует матрицу (кэш), размер которой можно рассчитать следующим образом:

$$M_{cache} = (n+1) \cdot (m+1) \cdot \text{size}(int) + \\ + \text{size}(int**) + (m+1) \cdot \text{size}(int*) \quad (4.4)$$

где $(n+1) \cdot (m+1)$ — количество элементов в кэше,
$\text{size}(int**)$ — размер указателя на матрицу,
$(m+1) \cdot \text{size}(int*)$ — размер указателя на строки матрицы.

Таким образом, затраты по памяти для рекурсивного алгоритма нахождения расстояния Дамерау — Левенштейна с использованием кэша:

$$M_{DLRecCache} = M_{DLRec} + M_{cache} \quad (4.5)$$

4.5 Вывод

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

1. При небольших длинах строк (длина < 5 симв.) разница между временем выполнения нерекурсивных реализаций алгоритмов Левенштейна и Дамерау — Левенштейна незначительна. Однако, при увеличении длины обрабатываемых строк алгоритм поиска расстояния Дамерау — Левенштейна выполняется на порядок дольше, что связано с обработкой дополнительного условия о перестановке символов.

2. Рекурсивная реализация алгоритма поиска расстояния Дамерау — Левенштейна с использованием кэша работает на порядок быстрее реализации поиска этого расстояния без кэширования.

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

В результате исследования алгоритмов по затрачиваемой памяти можно сделать вывод о том, что итеративные алгоритмы и рекурсивный алгоритм с кэшированием требуют больше памяти по сравнению с рекурсивным без оптимизаций. В реализациях, использующих матрицу, максимальный размер используемой памяти увеличивается пропорционально произведению длин строк, в то время как у рекурсивного алгоритма без кэширования объем затрачиваемой памяти увеличивается пропорционально сумме длин строк.

5 Заключение

В результате выполнения лабораторной работы для достижения этой цели были выполнены следующие задачи:

1. описаны алгоритмы поиска расстояния Левенштейна и Дамерау — Левенштейна;

2. разработаны и реализованы соответствующие алгоритмы;

3. создано программное обеспечение, позволяющее протестировать реализованные алгоритмы;

4. проведен сравнительный анализ процессорного времени выполнения реализованных алгоритмов:

   - при малых длинах строк (< 5) рекурсивные реализации с кешем и без для поиска расстояния Дамерау — Левенштейна имеют приблизительно одинаковое время работы, но с увеличением длины строки реализация без кеша выполняется на порядок дольше, поскольу не происходит повторное вычисление значений;
   - разница между итеративными реализациями алгоритмов поиска расстояний Левенштейна и Дамерау — Левенштейна незначительна, и обусловлена она дополнительным условием на проверку равенства соседних символов для расстояния Дамерау — Левенштейна;
   - итеративная реализация работает на порядок быстрее рекурсивной с кешем для поиска расстояния Дамерау — Левенштейна.

5. проведен сравнительный анализ затрачиваемой алгоритмами памяти: итеративные алгоритмы и рекурсивные алгоритмы с кешированием требуют больше памяти по сравнению с рекурсивным алгоритмом без кеширования. В реализациях, использующих матрицы, максимальный используемый объем памяти увеличивается пропорционально произведению длин строк. С другой стороны, для рекурсивного алгоритма без кеширования потребление памяти увеличивается пропорционально сумме длин строк.

Список использованных источников

1 А. Погорелов Д., М. Таразанов А. Сравнительный анализ алгоритмов редакционного расстояния Левенштейна и Дамерау-Левенштейна // Синергия Наук. 2019. URL: https://elibrary.ru/item.asp?id=36907767 (дата обращения 10.10.2023).

2 Документация по Microsoft C++ [Электронный ресурс]. — Режим доступа: https://learn.microsoft.com/ru-ru/cpp/?view=msvc-170&viewFallbackFrom=vs-2017 (дата обращения: 25.09.2023).

3 Standard library header <ctime> [Электронный ресурс]. — Режим доступа: https://en.cppreference.com/w/cpp/header/ctime.