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 «Анализы Алгоритмов»

на тему: «Редакционное расстояние»

Студент группы ИУ7-51Б ______ (Подпись, дата) Савинова М. Г. (Фамилия И.О.)

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

Преподаватель ______ (Подпись, дата) Строганов Ю. В.. (Фамилия И.О.)

Москва — 2023 г.

Содержание

Введение

Расстояние Левенштейна — это метрика, используемая для измерения разницы между двумя строками. Она вычисляет минимальное количество односимвольных изменений (вставок, удалений или замен), необходимых для преобразования одной строки в другую.

Расстояние Левенштейна обычно используется в различных областях, в том числе:

1) **Проверка орфографии:** выявление и исправление ошибок для слов на основе их расстояния Левенштейна.

2) **Анализ последовательности ДНК:** измерение сходства между последовательностями ДНК позволяет исследователям сравнивать и анализировать генетические данные.

3) **Обработка естественного языка (НЛП):** используется в таких задачах, как классификация текста, поиск информации и машинный перевод, для определения сходства между текстами.

Расстояние Дамерау-Левенштейна является расширением расстояния Левенштейна, которое не учитывает транспозиции. Дополнительная операция позволяет обрабатывать случаи, когда символы меняются местами или переупорядочиваются.

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

Необходимо выполнить следующие **задачи**:

1) изучить алгоритмы Левенштейна и Дамерау-Левенштейна для нахождения редакционного расстояния между строками;

2) реализовать данные алгоритмы;

3) выполненить сравнительный анализ алгоритмов по затрачиваемым ресурсам (времени, памяти);

4) описать и обосновать полученные результаты в отчете.

1 Аналитическая часть

1.1 Расстояние Левенштейна

Расстояние Левенштейна[1] — это минимальное количество редакторских операций вставки, замены и удаления, которые необходимо выполнить для преобразования одной строки в другую.

Каждая операция имеет свою цену(w). Редакционное предписание - это минимальная последовательность действий, которую необходимо выполнить для получения из первой строки вторую. Эта цена и есть искомое расстояние Левенштейна.

Введем следующие обозначения:

1) **I** (от англ. insert) - вставка ($w(\lambda, b) = 1$);

2) **R** (от англ. replace) - замена ($w(a, b) = 1$, $a \neq b$);

3) **D** (от англ. delete) - удаление ($w(a, \lambda) = 1$);

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

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

где сравнение символов строк S_1 и S_2 рассчитывается таким образом:

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

1.1.1 Нерекурсивный алгоритм нахождения расстояния Левенштейна

При больших M, N рекурсивная реализация алгоритма Левенштейна малоэффетивна **по времени** выполнения, поскольку промежуточные значения $D[i][j]$ вычисляются несколько раз. Для того, чтобы не производить одни и те же вычисления несколько раз, можно использовать матрицу для их хранения.

Матрица имеет следующие размеры:

$$(N+1) \times (M+1), \tag{1.3}$$

где M, N значения длин строк.

В ячейке $[i, j]$ хранится значение $D(S1[1...i], S2[1...j])$. Вся таблица заполняется в соответствии с формулой (1.1).

1.2 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна — это мера разницы двух строк, определяемая как минимальное количество операций вставки, удаления, замены и транспозиции (перестановки двух соседних символов), необходимых для перевода одной строки в другую. Является расширением расстояния Левенштейна, поскольку помимо трех базовых операций содержит еще операцию транспозиции T (от англ. transposition).

Расстояние Дамерау-Левенштейна определятся следующей рекуррентной формуле:

$$D(i,j) = \begin{cases} max(i,j), & \min(i,j) = 0, \\ min \begin{cases} D(i,j-1)+1, \\ D(i-1,j)+1, \\ D(i-1,j-1)+m(S_1[i],S_2[j]), \\ D(i-2,j-2)+1, \end{cases} & \begin{array}{l} \text{если } i,j > 1, \\ S_1[i] = S_2[j-1], \\ S_1[i-1] = S_2[j], \end{array} \\ min \begin{cases} D(i,j-1)+1, \\ D(i-1,j)+1, \\ D(i-1,j-1)+m(S_1[i],S_2[j]), \end{cases} & \text{иначе.} \end{cases} \tag{1.4}$$

1.2.1 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна

Каждый рекурсивный вызов соответствует одному из случаев:

- $D(i-1,j)+1$ соответствует удалению символа (из S_1 в S_2);
- $D(i,j-1)+1$ соответствует вставке символа (из S_1 в S_2);
- $D(i-1,j-1)+m(S_1[i],S_2[j]$ соответствие или несоответсвтие в зависимости от совпадения символов;
- $D(i-2,j-2)+1$ в случае перестановки двух последовательных символов;

1.2.2 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна с кешированием

При больших M, N рекурсивная реализация является малоэффективной **по времени**, поскольку приходится неоднократно считать одни и те же промежуточные значения. Для оптимизации по времени рекурсивного алгоритма нахождения расстояния Левенштейна можно использовать кеш, т.е. *хеш-таблицу*, состоящую из пар *«ключ» : «значение»*, где:

— *ключ* — вычисленное целочисленное значение;

— *значение* — $D[i, j]$;

1.2.3 Нерекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна

При больших значениях M, N рекурсивная реализация алгоритма Дамерау-Левенштейна малоэффетивна **по времени** выполнения, поскольку промежуточные значения $D[i][j]$ вычисляются несколько раз. Для того, чтобы не производить одни и те же вычисления несколько раз, можно использовать матрицу для их хранения.

Матрица имеет следующие размеры:

$$(N + 1) \times (M + 1), \tag{1.5}$$

где M, N значения длин строк.

В ячейке $[i, j]$ хранится значение $D(S1[1...i], S2[1...j])$. Вся таблица заполняется в соответствии с формулой (1.4).

Вывод

Формулы для вычисления расстояний Левенштейна и Дамерау-Левенштейна задаются **рекуррентно**, поэтому алгоритмы для нахождения их расстояний можно реализовать как *итеративно*, так и *рекурсивно*.

2 Конструкторская часть

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

2.1 Требования к программному обеспечению

К программе предъявлен ряд требований:

— наличие интерфейса для выбора действий;

— возможность ввода строк;

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

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

2.2 Требования к вводу

1) На вход подаются две строки.

2) Буква нижнего и верхнего регистра считаются разными символами.

3) Строки могут включать как символы латиницы, так и кириллицы.

2.3 Разработка алгоритмов

На вход алгоритмов подаются строки S_1 и S_2.

На рисунке 2.1 представлена схема алгоритма поиска расстояния Левенштейна. На рисунках 2.2 – 2.4 представлены схемы алгоритмов поиска Дамерау-Левенштейна.

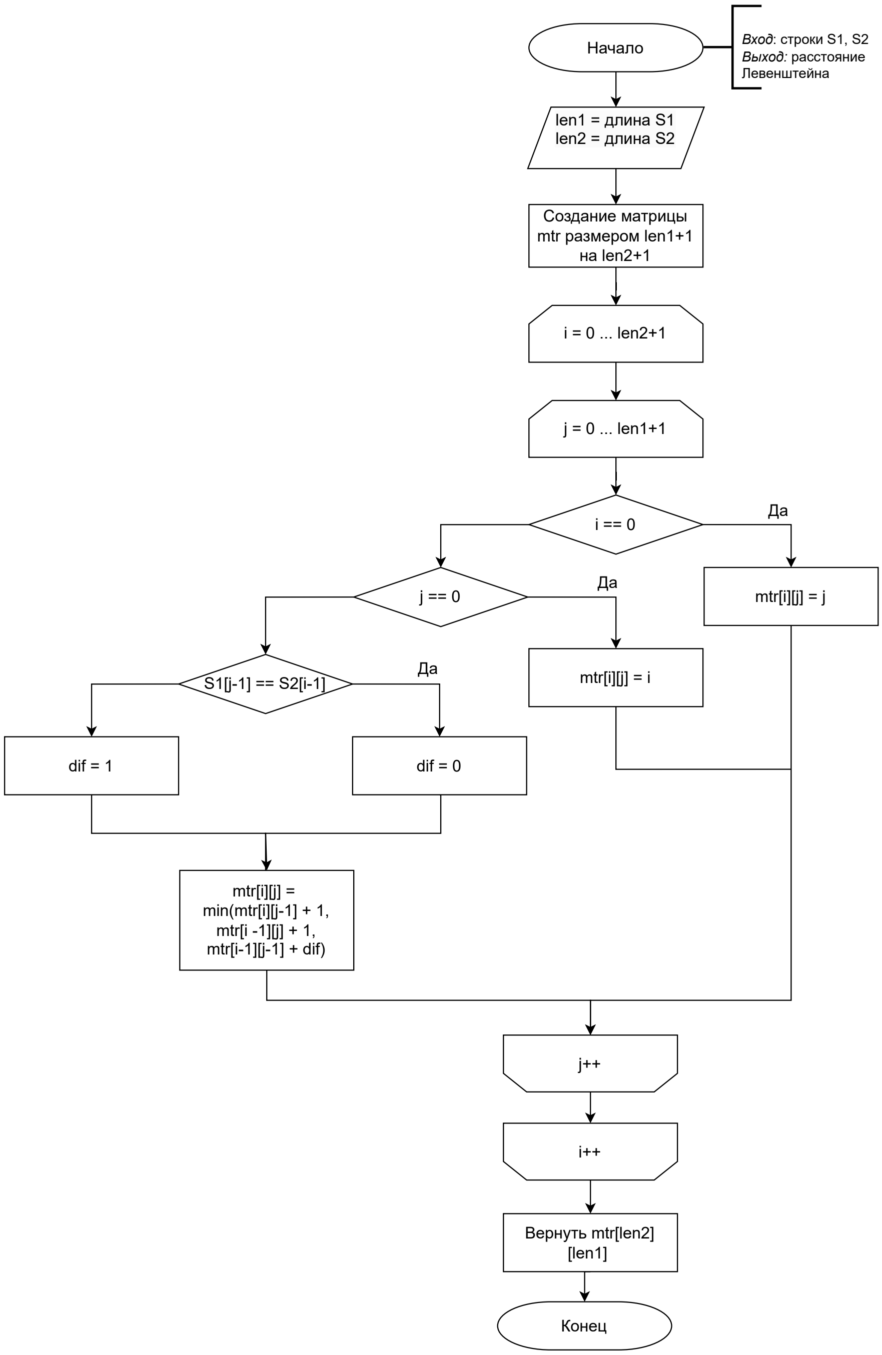

Рисунок 2.1 – Схема нерекурсивного алгоритма нахождения расстояния Левенштейна

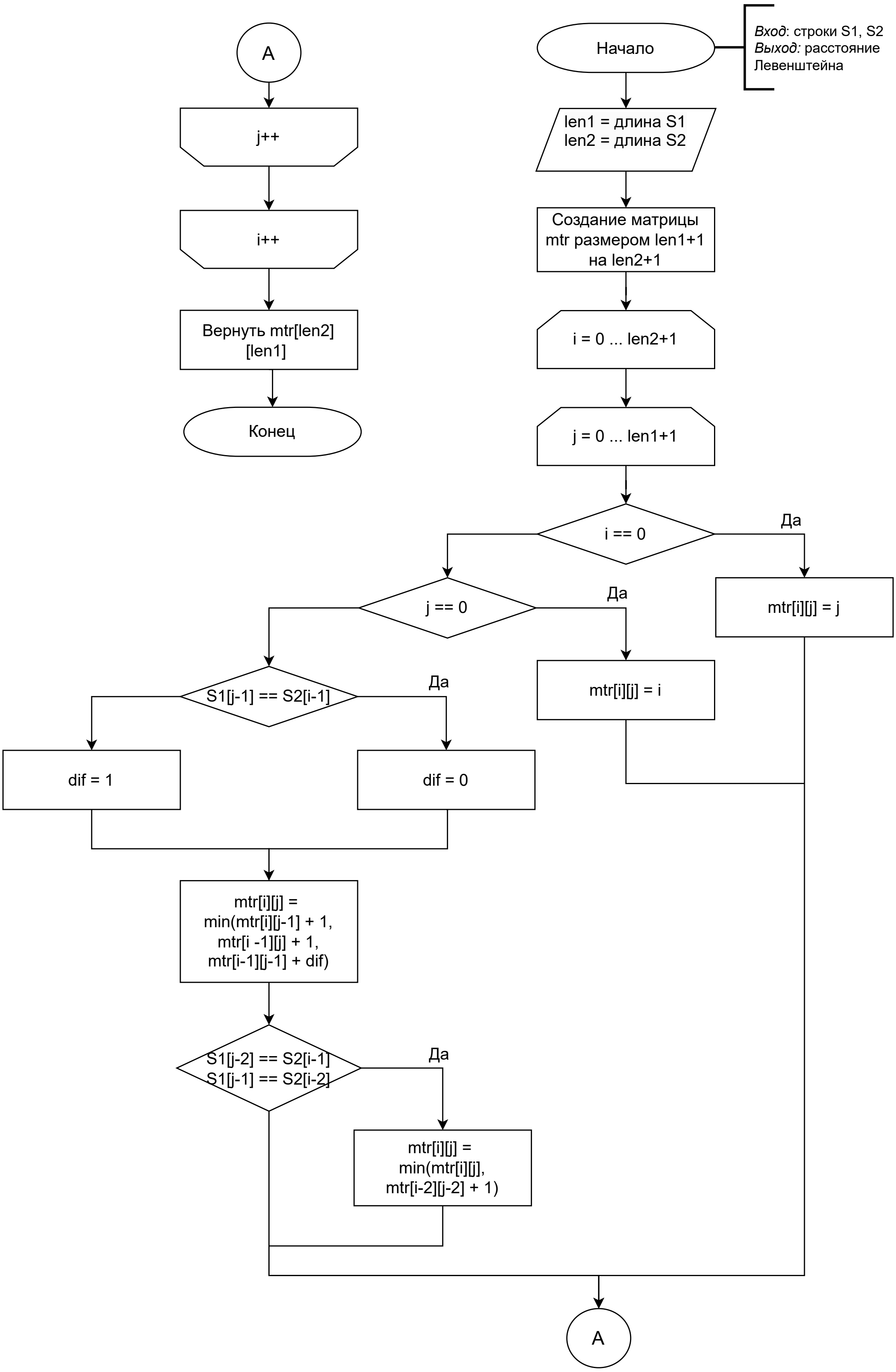

Рисунок 2.2 – Схема нерекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

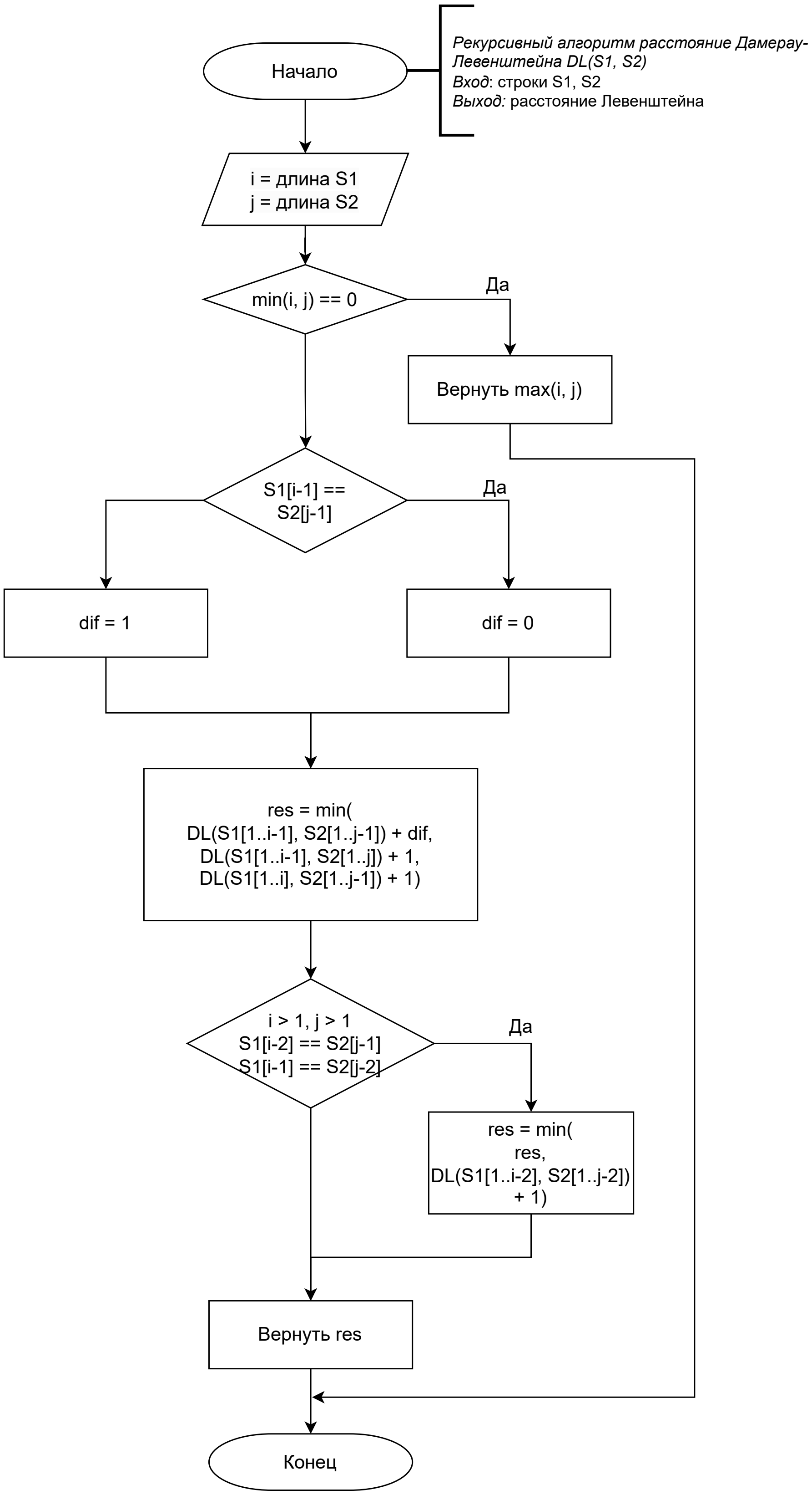

Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

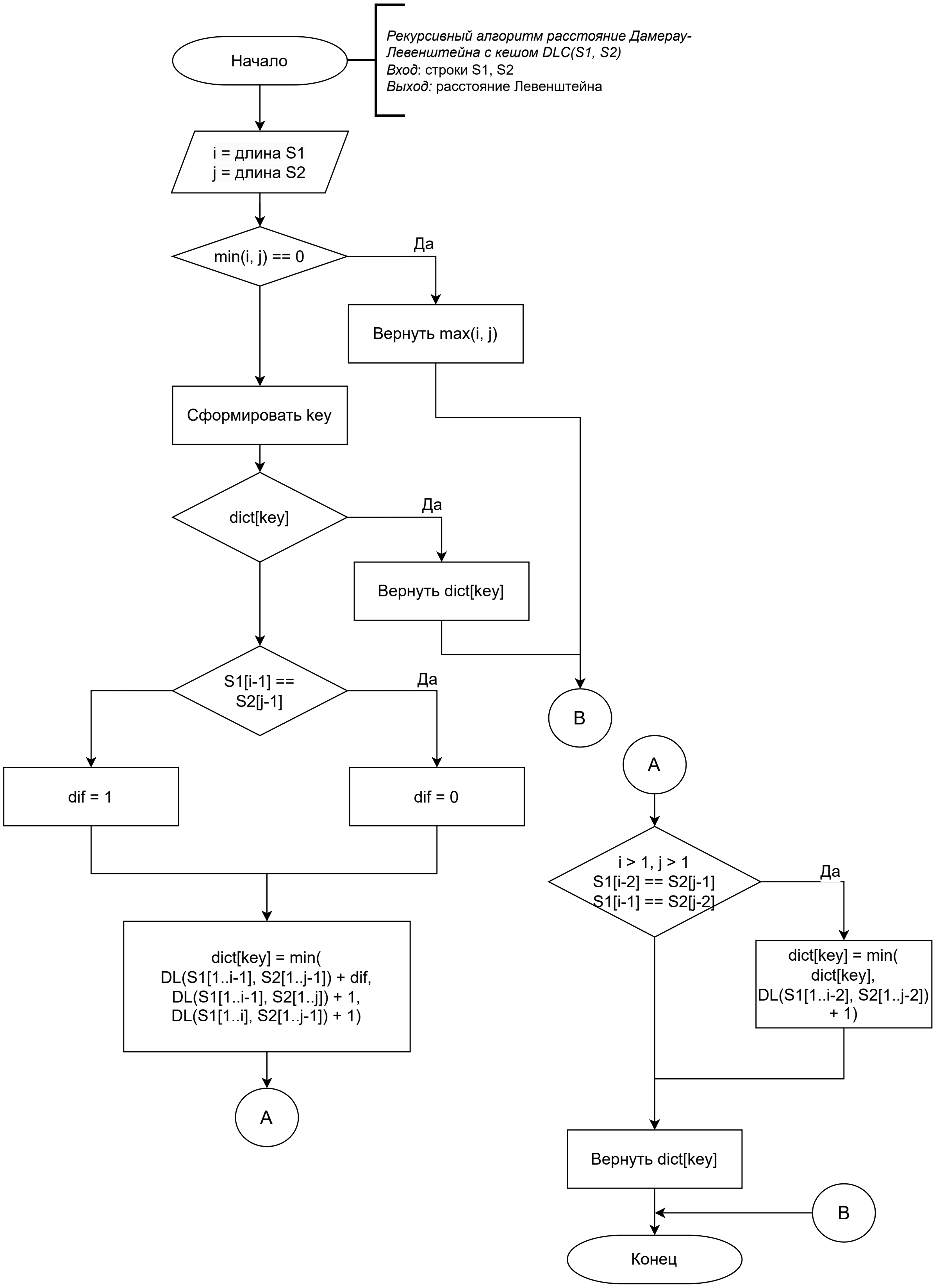

Рисунок 2.4 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с кешированием

2.4 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие структуры данных:

- *строка* — массив типа $wchar_t$ размером длины строки;
- *длина строки* — целое число типа int;
- *матрица* — двумерный массив значений типа int.
- *хеш-таблицы* — контейнер $std :: map$, где ключ - int, значение - int.

Вывод

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

3 Технологическая часть

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

3.1 Средства реализации

Для реализации данной лабораторной работы был выбран язык $C{+}{+}$ [2], так как в нем есть стандартная библиотека *chrono*, которая позволяет произвести замеры процессорного времени выполнения программы, а так же наличие типа данных $std :: wstring$, позволяющего хранить как кириллические символы, так и латинские; В качестве среды разработки был выбран $VisualStudioCode$: он является кроссплатформенным и предоставляет полный функционал для проектирования и отладки кода.

3.2 Сведения о модулях программы

Данная программа разбита на следующие модули.

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

— `algorithms.cpp` — файл содержит функции поиска расстояния Левенштейна и Дамерау-Левенштейна.

— `matrix.cpp` — файл содержит функции динамического выделения и очищения памяти для матрицы, а так же ее вывод на экран.

— `measure.cpp` — файл содержит функции, замеряющее процессорное время для алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна.

3.3 Реализация алгоритмов

В листингах 3.2 – 3.6 приведены реализации алгоритмов поиска расстояний Левенштейна (только нерекурсивный алгоритм) и Дамерау-Левенштейна (нерекурсивный, рекурсивный и рекурсивный с кешированием).

Листинг 3.1 – Функция нахождения расстояния Левенштейна с использованием матрицы (начало)

```
int notRecursiveLev(wstring &word1, wstring &word2, bool print) {

    int len1 = word1.length();
    int len2 = word2.length();

    int** mtr = allocateMtr(len2 + 1, len1 + 1);

    if (!mtr)
        return 0;

    for (int i = 0; i <= len2; ++i) {

        for (int j = 0; j <= len1; ++j) {

            if (i == 0)
                mtr[i][j] = j;
            else if (j == 0)
                mtr[i][j] = i;
            else {
                int dif = (word1[j - 1] == word2[i - 1]) ? 0 : 1;

                mtr[i][j] = min(mtr[i - 1][j] + 1,
                                min(mtr[i][j - 1] + 1, mtr[i - 1][j - 1] + dif));
            }
        }
    }
```

Листинг 3.2 – Функция нахождения расстояния Левенштейна с использованием матрицы (окончание)

```
    if (print)
        printMtr(mtr, word1, word2);

    int res = mtr[len2][len1];
    freeMtr(mtr, len2 + 1);

    return res;
}
```

Листинг 3.3 – Функция нахождения расстояния Дамерау-Левенштейна с использованием матрицы (начало)

```
int notRecursiveDamLev(wstring &word1, wstring &word2, bool
    print) {

    int len1 = word1.length();
    int len2 = word2.length();

    int** mtr = allocateMtr(len2 + 1, len1 + 1);

    if (!mtr)
        return 0;

    for (int i = 0; i <= len2; ++i) {

        for (int j = 0; j <= len1; ++j) {

            if (i == 0)
                mtr[i][j] = j;
            else if (j == 0)
                mtr[i][j] = i;
            else {
                int dif = (word1[j - 1] == word2[i - 1]) ? 0 :
                    1;

                mtr[i][j] = min(mtr[i - 1][j] + 1,
                                min(mtr[i][j - 1] + 1, mtr[i -
                                    1][j - 1] + dif));
```

Листинг 3.4 – Функция нахождения расстояния Дамерау-Левенштейна с использованием матрицы (окончание)

```
                if (word1[j - 2] == word2[i - 1] && word1[j - 1] == word2[i - 2])
                    mtr[i][j] = min(mtr[i][j], mtr[i - 2][j - 2] + 1);
            }
        }
    }

    if (print)
        printMtr(mtr, word1, word2);

    int res = mtr[len2][len1];
    freeMtr(mtr, len2 + 1);

    return res;
}
```

Листинг 3.5 – Функция нахождения расстояния Дамерау-Левенштейна рекурсивно

```
int recursive(wstring &word1, wstring &word2, int ind1, int
    ind2) {

    if (min(ind1, ind2) == 0)
        return max(ind1, ind2);

    int dif = (word1[ind1 - 1] == word2[ind2 - 1]) ? 0 : 1;

    int res = min(recursive(word1, word2, ind1 - 1, ind2 - 1) +
        dif,
                  min(recursive(word1, word2, ind1 - 1, ind2) +
                     1,
                      recursive(word1, word2, ind1, ind2 - 1) +
                         1));

    if (word1[ind1 - 1] == word2[ind2 - 2] && word1[ind1 - 2]
        == word2[ind2 - 1])
        res = min(res, recursive(word1, word2, ind1 - 2, ind2 -
            2) + 1);

    return res;
}
```

Листинг 3.6 – Функция нахождения расстояния Дамерау-Левенштейна рекурсивно с кешированием

```
int recursiveCash(wstring &word1, wstring &word2, int ind1, int
    ind2, mapT& dict) {

    int key = ind1 + ind2 * (1 + word2.length());

    if (dict[key])
        return dict[key];

    if (min(ind1, ind2) == 0)
        return dict[key] = max(ind1, ind2);

    int dif = (word1[ind1 - 1] == word2[ind2 - 1]) ? 0 : 1;

    int res = min(recursiveCash(word1, word2, ind1 - 1, ind2 -
        1, dict) + dif,
                    min(recursiveCash(word1, word2, ind1 - 1,
                        ind2, dict) + 1,
                        recursiveCash(word1, word2, ind1, ind2 -
                            1, dict) + 1));

    if (ind1 > 1 && ind2 > 1 && word1[ind1 - 1] == word2[ind2 -
        2] && word1[ind1 - 2] == word2[ind2 - 1])
        res = min(res, recursiveCash(word1, word2, ind1 - 2,
            ind2 - 2, dict) + 1);

    dict[key] = res;

    return dict[key];
}
```

3.4 Функциональные тесты

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

Таблица 3.1 – Функциональные тесты

<table>
<tr><th colspan="2">Входные данные</th><th colspan="4">Расстояние и алгоритм</th></tr>
<tr><th rowspan="3">Строка 1</th><th rowspan="3">Строка 2</th><th>Левенштейна</th><th colspan="3">Дамерау-Левенштейна</th></tr>
<tr><th rowspan="2">Итеративный</th><th rowspan="2">Итеративный</th><th colspan="2">Рекурсивный</th></tr>
<tr><th>Без кеша</th><th>С кешом</th></tr>
<tr><td>a</td><td>b</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td>a</td><td>a</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>кот</td><td>скат</td><td>2</td><td>2</td><td>2</td><td>2</td></tr>
<tr><td>кот</td><td>кто</td><td>2</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td>Австралия</td><td>Австрия</td><td>2</td><td>2</td><td>2</td><td>2</td></tr>
<tr><td>кот</td><td>ток</td><td>2</td><td>2</td><td>2</td><td>2</td></tr>
<tr><td>слон</td><td>слоны</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
</table>

Вывод

Были реализованы и протестированы алгоритмы поиска расстояния Левенштейна итеративно, а также поиска расстояния Дамерау–Левенштейна итеративно, рекурсивно и рекурсивного с кеширования. Проведено тестирование реализаций алгоритмов.

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры по времени, представлены далее.

— Процессор: AMD Ryzen 5 5500U with Radeon Graphics 2.10 GHz.

— Оперативная память: 16 ГБайт.

— Операционная система: Windows 10 Pro 64-разрядная система версии 22H2 [3].

При замерах времени ноутбук был включен в сеть электропитания и был нагружен только системными приложениями.

4.2 Демонстрация работы программы

На рисунке 4.1 представлена демонстрация работы разработанного ПО, а именно показаны результаты работы алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна на примере двух строк *«секста»* и *«септима»*.

```
                Меню
1. Запуск алгоритмов поиска расстояния Левенштейна:
   1) Нерекурсивный Левенштейна;
   2) Нерекурсивный Дамерау-Левенштейна;
   3) Рекурсивный Дамерау-Левенштейна без кэша;
   4) Рекурсивный Дамерау-Левенштейна с кэшом;
2. Замерить время и память для реализованных алгоритмов;
0. Выход

Выберете пункт (0-2): 1

Введите 1е слово: секста
Введите 2е слово: септима

Минимальное кол-во операций:

              с  е  к  с  т  а
           0  1  2  3  4  5  6
        с  1  0  1  2  3  4  5
        е  2  1  0  1  2  3  4
        п  3  2  1  1  2  3  4
        т  4  3  2  2  2  2  3
        и  5  4  3  3  3  3  3
        м  6  5  4  4  4  4  4
        а  7  6  5  5  5  5  4
   1) Нерекурсивный Левенштейна:                4

              с  е  к  с  т  а
           0  1  2  3  4  5  6
        с  1  0  1  2  3  4  5
        е  2  1  0  1  2  3  4
        п  3  2  1  1  2  3  4
        т  4  3  2  2  2  2  3
        и  5  4  3  3  3  3  3
        м  6  5  4  4  4  4  4
        а  7  6  5  5  5  5  4
   2) Нерекурсивный Дамерау-Левенштейна:        4
   3) Рекурсивный Дамерау-Левенштейна без кэша: 4
   4) Рекурсивный Дамерау-Левенштейна с кэшом:  4
```

Рисунок 4.1 – Демонстрация работы программы

4.3 Временные характеристики

Все реализованный алгоритмы сравнивались на случайно сгенерированных строках длиной от 0 до 10 с шагом 1, а нерекурсивные алгоритмы сравнивались еще на строках длиной от 20 до 100 с шагом 10. Поскольку замеры по времени имеют некоторую погрешность, для каждой строки и каждой реализации алгоритма замеры производились 10 раз, а затем вычислялось среднее арифметическое значение.

На рисунке 4.3 представлен график, иллюстрирующий зависимость времени работы от длины строк для матричных реализаций алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна. Время их работы растет соизмеримо, но алгоритм поиска расстояния Дамерау-Левенштейна работает дольше, так как в нем присутствует дополнительная проверка для обмена соседних символов.

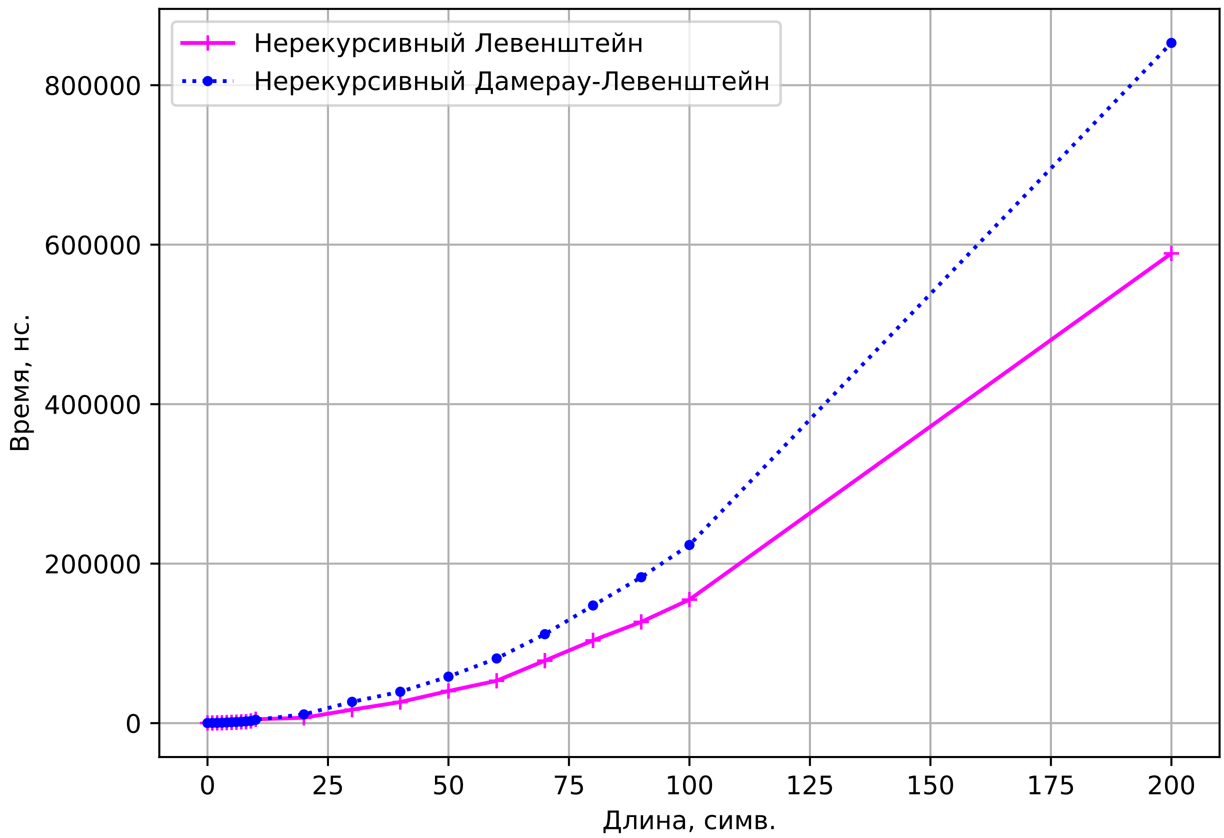

Рисунок 4.2 – Сравнение по времени матричных реализаций алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна

На рисунке 4.3 представлен график, иллюстрирующий зависимость времени работы от длины строк для рекурсивных реализаций алгоритмов поиска расстояния Дамерау-Левенштейна с кешом и без.

Рисунок 4.3 – Сравнение по времени рекурсивных реализаций алгоритмов поиска расстояния Дамерау-Левенштейна с кешом и без

На рисунке 4.4 представлен график, иллюстрирующий зависимость времени работы от длины строк для матричных реализаций. На графике видно, что матричные реализации работают значительно быстрее рекурсивной реализации с кешированием.

4.4 Характеристики по памяти

Введем следующие обозначения:

— n — длина строки S_1;

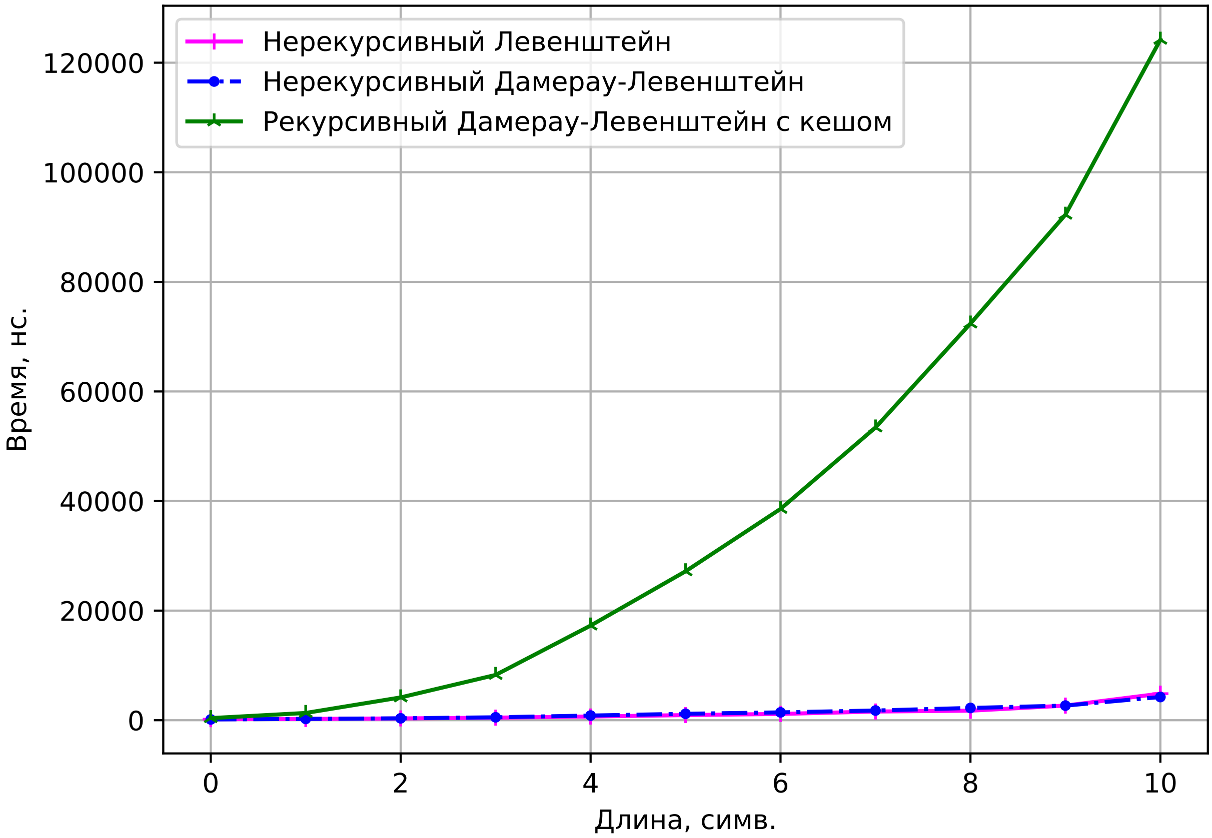

Рисунок 4.4 – Сравнение по времени матричных реализаций алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна и рекурсивной реализации с кешом поиска расстояния Дамерау-Левенштейна

— m — длина строки S_2;

— $size()$ — функция вычисляющая размер в байтах;

— $char$ — тип, используемый для хранения символа строки;

— int — целочисленный тип;

Использование памяти при **итеративной реализации** алгоритма поиска расстояния Левенштейна теоретически равно:

$$\begin{aligned} M_{iter} = (n+1) \cdot (m+1) \cdot size(int) + (n+m) \cdot size(char) + \\ +5 \cdot size(int) + size(int**) + (n+1) \cdot size(int*), \end{aligned} \tag{4.1}$$

где:

— $(n+1) \cdot (m+1) \cdot size(int)$ — хранение матрицы;

— $(n+m) \cdot size(char)$ — хранение двух строк;

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

— $3 \cdot size(int)$ — дополнительные переменные;

— $size(int**) + (n+1) \cdot size(int*)$ — указатель на матрицу;

Использование памяти при **итеративной реализации** алгоритма поиска расстояния Дамерау-Левенштейна идентично формуле 4.1

Рассчитаем затраты по памяти для **рекурсивного** алгоритма поиска расстояния Дамерау-Левенштейна *(для каждого вызова)*:

$$M_{call} = (n+m) \cdot size(char) + 4 \cdot size(int) + 8 \tag{4.2}$$

где:

— $(n+m) \cdot size(char)$ — хранение двух строк;

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

— $2 \cdot size(int)$ — дополнительные переменные;

— 8 байт — адрес возврата;

Макисмальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящий строк, поэтому макисмальный расход памяти равен:

$$M_{rec} = (n+m) \cdot M_{call} \tag{4.3}$$

где:

— $n+m$ — максимальная глубина стека;

— M_{call} — затраты по памяти для одного рекурсивного вызова;

Для рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна с использованием кеша необходимо подсчитать размер самого кеша:

$$M_{cash} = (m+1) \cdot (n+1) \cdot (size(int) + size(int)) \tag{4.4}$$

где:

— $(m + 1) \cdot (n + 1)$ — количество элементов в кеше;

— $size(int)$ — хранение значения;

— $size(int)$ — хранение ключа;

Затраты по памяти для рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна с учетом кеша:

$$M_{recCash} = M_{rec} + M_{cash} \tag{4.5}$$

4.5 Вывод

Алгоритм нахождения расстояния Дамерау-Левенштейна по времени выполнения незначительно отличается от алгоритма нахождения расстояния Левенштейна.

Рекурсивные реализации поиска расстояния Дамерау-Левенштейна работают на порядок дольше матричной реализации.

По расходу памяти матричный алгоритм проигрывает рекурсивному: макисмальный размер используемой памяти у матричного алгоритма растет как *произведение* длиин строк, а то время как у рекурсивного - как *сумма* длин строк.

Заключение

В результате выполнения лабораторной работы при иcсследовании алгоритмов нахождения расстояний Левенштейна и Дамерау-Левегштейна были выполнены следующие задачи:

1) Описаны алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштей

2) Разработаны и реализованы соответствующие алгоритмы;

3) Создан программный продукт, позволяющий протестировать реализованные алгоритмы;

4) Проведен сравнительный анализ процессорного времени выполнения реализованныйх алгоритмов;

5) Проведен сравнительный анализ затрачиваемой алгоритмами памяти;

Список использованных источников

1 И. Левенштейн В. Двоичные коды с исправлением выпадений, вставок и замещений символов. — М.: Издательство «Наука», Доклады АН СССР, 1965. Т. 163.

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