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 (ИУ)
КАФЕДРА Программное обеспечение ЭВМ и информационные технологии (ИУ7)

Отчет по лабораторной работе №1
По курсу: «Анализ алгоритмов»

«*Алгоритм по нахождению расстояния Левенштейна и Дамерау-Левенштейна*»

Выполнил студент: ***Тимонин Антон Сергеевич***
фамилия, имя, отчество

Группа: ***ИУ7-52Б***

Преподаватель:. **Волкова Л.Л.**
подпись, дата

Оценка _____ Дата _____

2019 г.

Оглавление

Введение 3

1 Аналитическая часть 4

1.1 Описание алгоритмов 4

1.2 Расстояние Левенштейна 5

1.3 Расстояние Дамерау-Левенштейна 5

2 Конструкторская часть 7

2.1 Требования к программному продукту 7

2.2 Схемы алгоритмов 8

2.3 Сравнительный анализ 13

3 Технологическая часть 14

3.1 Выбор средств реализации 14

3.2 Листинг кода 14

4 Экспериментальная часть 17

4.1 Примеры работы программы 17

4.2 Постановка эксперимента 17

4.3 Сравнительный анализ 19

Заключение 21

Введение

Целью данной лабораторной работы является изучение метода динамического программирования на материале алгоритмов Левенштейна и Левенштейна-Дамерау.

Задачами данной лабораторной работы являются:

1. реализация матричного способа нахождения обоих расстояний и рекурсивного способа нахождения расстояния Дамерау-Левенштейна;
2. применение метода динамического программирования для матричной реализации указанных алгоритмов;
3. получение практических навыков реализации указанных алгоритмов;
4. сравнительный анализ линейной и рекурсивной реализации выбранного алгоритма;
5. описание и обоснование полученных результатов в отчете о выполненной лабораторной работе.

1 Аналитическая часть

В данном разделе приведены описания расстояний Левенштейна и Дамерау-Левенштейна

1.1 Описание алгоритмов

Редакционное расстояние (*дистанция редактирования*) — минимальное количество операций вставки одного символа **I**, удаления одного символа **D** и замены одного символа на другой **R**, необходимых для превращения одной строки в другую. Измеряется для двух строк, часто используется в компьютерной лингвистике и биоинформатике, а в частности:

- для исправления ошибок в слове (в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи);
- для сравнения текстовых файлов утилитой diff и ей подобными. Здесь роль «символов» играют строки, а роль «строк» — файлы;
- в биоинформатике для сравнения генов, хромосом и белков.

В общем случае операции выглядят следующим образом:

- D(a, b) – цена замены символа a на b
- D(ø, b) – цена вставки символа b
- D(a, ø) – цена удаления символа a

Также необходимо найти последовательность замен, минимизирующую суммарную цену:

Пусть a, b – символы; Ɛ - пустой символ;

- w(a, a) = 0;
- w(a, b) = 1 при a≠b
- w(a, Ɛ) = 1
- w(Ɛ, a) = 1

1.2 Расстояние Левенштейна

Расстояние Левенштейна – это мера разницы двух строк символов, определяемая как минимальное количество операций вставки, удаления, замены.

Классификация операций и штрафов (на выполнение операции) в алгоритме Левенштейна:

1. Вставка символа I (*insert) = 1*
2. Удаление символа D (*delete*) = 1
3. Замена символа R (*replace*) = 1
4. Совпадение символа M (*match*) = 0

На вход алгоритма подаются две строки s1, s2 и их длины соответственно len1, len2. Тогда расстояние Левенштейна можно представить по рекуррентной формуле:

$$D(s1[1..len1], s2[1..len2]) = min(D(s1[1..len1 - 1], s2[1..len2]) + 1,$$
$$s1[1..len1], s2[1..len2 - 1]+1,$$
$$s1[1..len1 - 1], s2[1..len2 - 1] + \alpha),$$

Где $\alpha = 0$, если s1[len1] = s2[len2], иначе $\alpha = 1$

1.3 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-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

По сути, это алгоритм является модификацией алгоритма расстояния Левенштейна, в который добавили операцию транспозиции.

Транспозиции необходима, на случай, если 2 символа стоят не в том порядке:

1. Транспозиция T (*transposition*) = 1

А также в рекуррентную формула добавляется еще один член:

$$D(s1[1..len1], s2[1..len2 - 1]) + \beta,$$

где $\beta = 1$, если $s1[len1 - 1] = s2[len2]$ и $s2[len2 - 1] = s1[len1]$, а иначе $\beta = 0$

тоговую формулу

2 Конструкторская часть

В разделе конструкторская часть приведены требования к программе, математическое описание алгоритмов, блок схемы, сравнительный анализ

2.1 Требования к программному продукту

Требования в вводу:

1. На вход подаются буквы латинского алфавита.

Требования к программе:

1. работа программы без аварийных завершений программы.

2.2 Схемы алгоритмов

Матричная реализация алгоритма нахождения расстояния Левенштейна приведена на рис. 1.

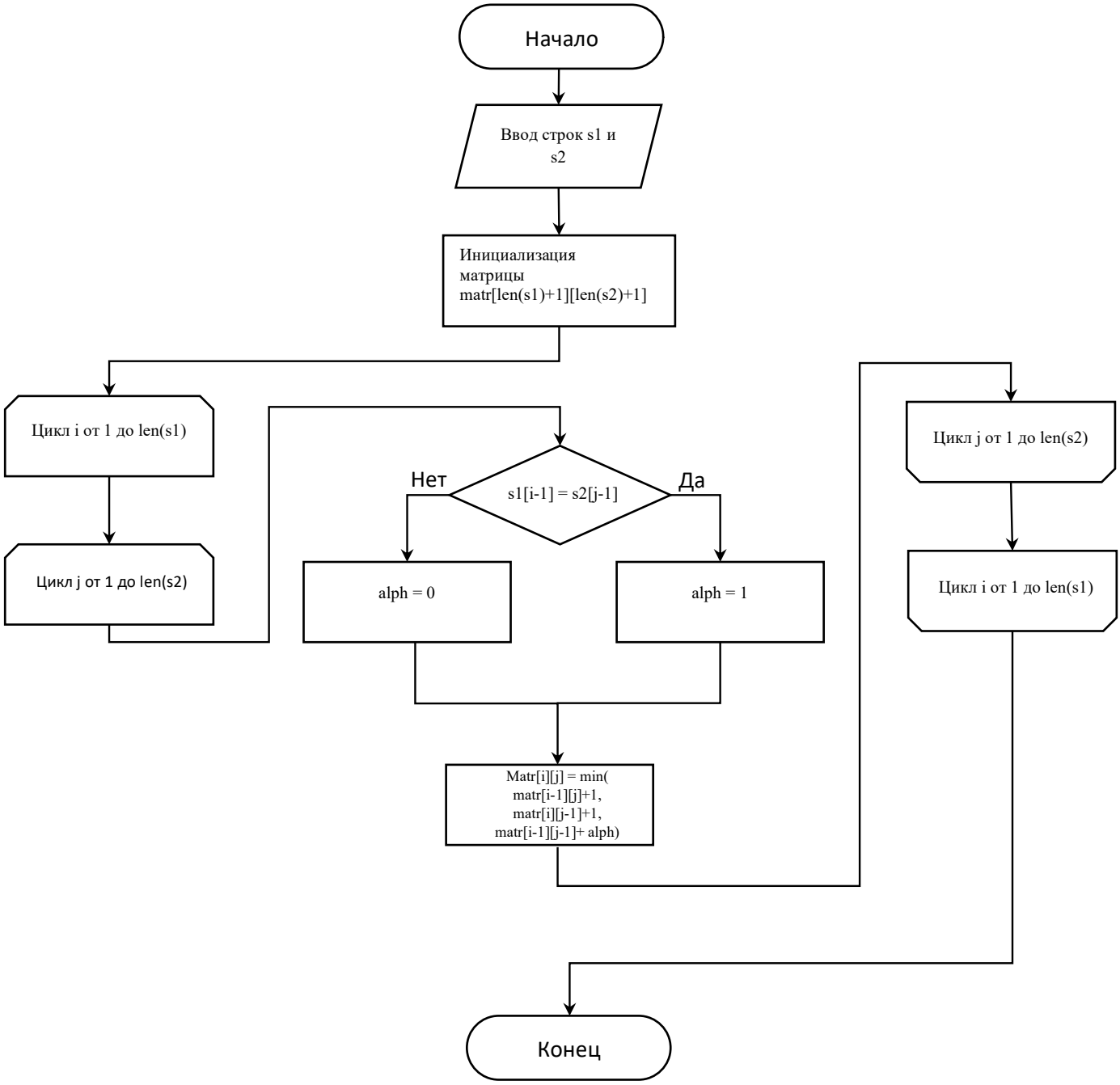

Рис. 1 – схема итеративного алгоритма поиска расстояния Левенштейна

Рекурсивная реализация алгоритма нахождения расстояния Дамерау-Левенштейна (Рис. 2)

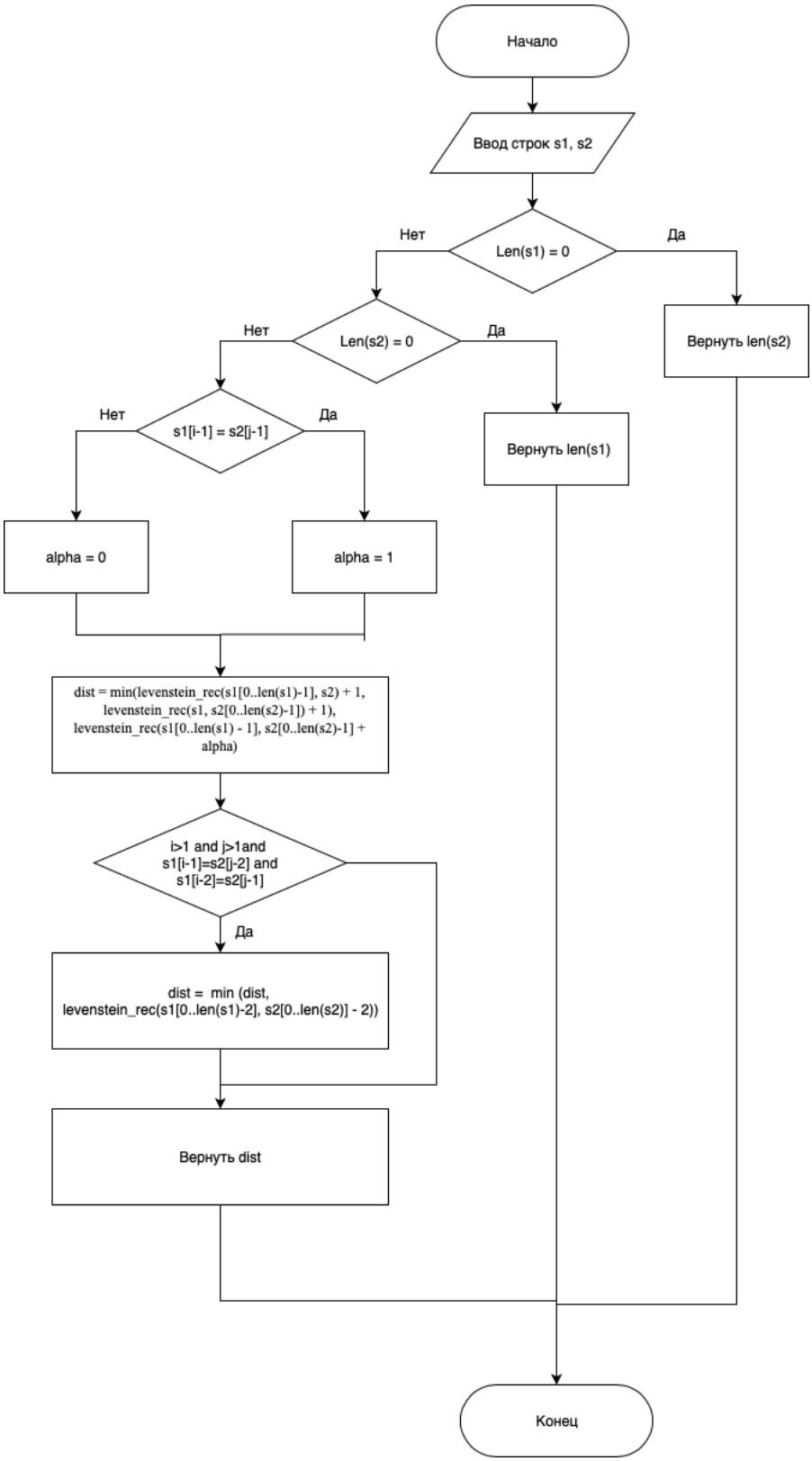

Рис.2 – схема рекурсивного алгоритма нахождения расстояние Дамерау-Левенштейна

Матричная реализация алгоритма нахождения расстояния Дамерау-Левенштейна (Рис. 3)

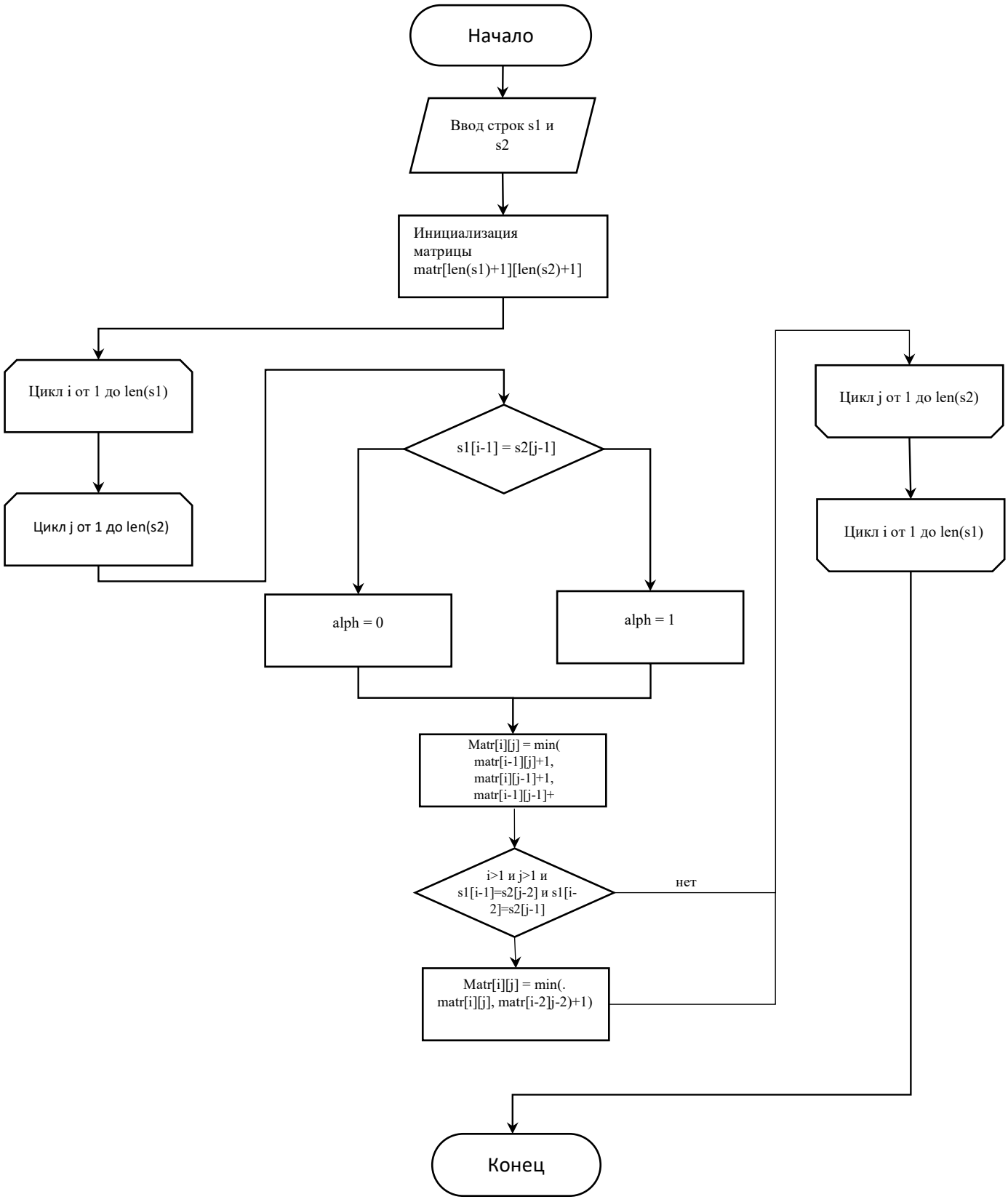

Рис. 3 – схема итеративного алгоритма нахождения расстояния Дамерау-Левенштейна

2.3 Сравнительный анализ

При больших входных данных рекурсивная реализация занимает значительно больше тиков процессора, чем обычная, матричная реализация. Но в некоторых случаях, когда длина слов, подающихся на вход, мала, тогда рекурсивная реализация занимает меньше памяти.

3 Технологическая часть

В данном разделе приведено требования к программному обеспечению, а также листинг кода

3.1 Выбор средств реализации

Программа была написана на языке C++ в среде QtCreator. Данный язык обусловлен наличием библиотеки, позволяющей считать тики процессора, что более точно определяет эффективность программы. Для измерения времени работы алгоритма используется библиотека “ctime”.

3.2 Листинг кода

Матричная реализация алгоритма Левенштейна представлена в листинге 1.

Листинг 1. Код итеративной реализации алгоритма Левенштейна

```
int levenstein(string s1, string s2)
{
    int len_s1 = s1.length() + 1;
    int len_s2 = s2.length() + 1;

    int arr[len_s1][len_s2];

    for (int i = 0; i < len_s1; i++) {
        for (int j = 0; j < len_s2; j++) {
            if (i * j == 0) {
                arr[i][j] = i + j;
            } else {
                arr[i][j] = 0;

            }
            cout << arr[i][j] << " ";
        }
        cout << "\n";
    }

    for (int i = 1; i < len_s1; i++) {
        for (int j = 1; j < len_s2; j++) {
            int key = 1;
            if (s1[i-1] == s2[j-1]) {
                key = 0;
            }
             arr[i][j] = Mmin(arr[i-1][j]+1, arr[i][j-1]+1,
                                             arr[i-1][j-1]+key);

        }
    }

    return arr[len_s1 - 1][len_s2 - 1];
}
```

Матричная реализация алгоритма Дамерау-Левенштейна представлена в листинге 2.

Листинг 2. Код итеративной реализации алгоритма Дамерау-Левенштейна

```
int dameray_levenstein(string s1, string s2)
{

    int len_s1 = s1.length() + 1;
    int len_s2 = s2.length() + 1;
    int key = 1;

    int arr[len_s1][len_s2];

    for (int i = 0; i < len_s1; i++) {
        for (int j = 0; j < len_s2; j++) {
            if (i * j == 0) {
                arr[i][j] = i + j;
            } else {
                arr[i][j] = 0;
            }
        }
    }

    for (int i = 1; i < len_s1; i++) {
        for (int j = 1; j < len_s2; j++) {
            key = 1;

            if (s1[i-1] == s2[j-1])
                key = 0;

            arr[i][j] = Mmin(arr[i-1][j] + 1, arr[i][j-1] + 1,
                                              arr[i-1][j-1] + key);

            if (i > 1 && j > 1 && s1[i-1] == s2[j-2] && s1[i-2] == s2[j-1]) {
                arr[i][j] = Mmin(arr[i][j], arr[i][j], arr[i-2][j-2] + 1);
            }

        }
    }
    return arr[len_s1 - 1][len_s2 - 1];
}
```

Рекурсивная реализация алгоритма поиска расстояния Дамерау-Левенштейна

Листинг 3. Код рекурсивной реализации алгоритма Дамерау-Левенштейна

```
int dameray_levenstein_rec(string s1, string s2)
{
    int var = 1;
    int dist = 0;
    int s1_l, s2_l;

    s1_l = Llen(s1);
    s2_l = Llen(s2);
```

```
    if (s1 == "" || s2 == "") {
        dist = max(s1.length(), s2.length());
        return dist;
    }

    string s1_new = s1.substr(0, s1.length() - 1);
    string s2_new = s2.substr(0, s2.length() - 1);

    if (s1[s1_l] == s2[s2_l]) {
        var = 0;
    }

    dist = Mmin (dameray_levenstein_rec(s1_new, s2) + 1,
                 dameray_levenstein_rec(s1, s2_new) + 1,
                 dameray_levenstein_rec(s1_new, s2_new) + var);

    if (s1.length() >= 2 && s2.length() >= 2 && s1[s1_l] == s2[s2_l - 1] &&
            s1[s1_l - 1] == s2[s2_l]) {
        string s1_damer = s1.substr(0, s1.length() - 2);
        string s2_damer = s2.substr(0, s2.length() - 2);
        dist = std::min(dist,dameray_levenstein_rec(s1_damer, s2_damer) + 1);
    }

    return dist;
}
```

4 Экспериментальная часть

В данном приведены тесты на корректную работу реализованных алгоритмов. А также проведены замеры по времени(в тиках процессора) и замеры по памяти.

4.1 Примеры работы программы

Пример работы программы представлены в таблицах 1.1 и 1.2. В 3-4 столбцах приведены результирующие расстояния, алгоритмы идут в порядке: матричный алгоритм расстояние Левенштейна, рекурсивный алгоритм Левенштейна, матричный алгоритм Дамерау-Левенштейна.

Таблица 1.1 Пример работы алгоритмов Левенштейна и Дамерау-Левенштейна

Слово №1	Слово №2	Вывод	Ожидаемый вывод
mainframe	aminframe	2 1 1	2 1 1
mainframe	mainfrem	9 8 8	9 8 8
mainframe		9 9 9	9 9 9
mainframe	mainframe	0 0 0	0 0 0

Таблица 1.2 Пример работы алгоритмов Левенштейна и Дамерау-Левенштейна

Слово №1	Слово №2	Вывод	Ожидаемый вывод
abcdef	badcfe	4 3 3	4 3 3
abcdef	abcfed	2 2 2	2 2 2
abcdef	cbed	4 4 4	4 4 4
abcdef	fedcba	6 5 5	6 5 5

4.2 Постановка эксперимента

Оценка эффективности алгоритмов по скорости работы представлена на рис. 4 и рис. 5.

Рис. 4 – Эффективность по времени работы в зависимости от входных слов длиной до 10

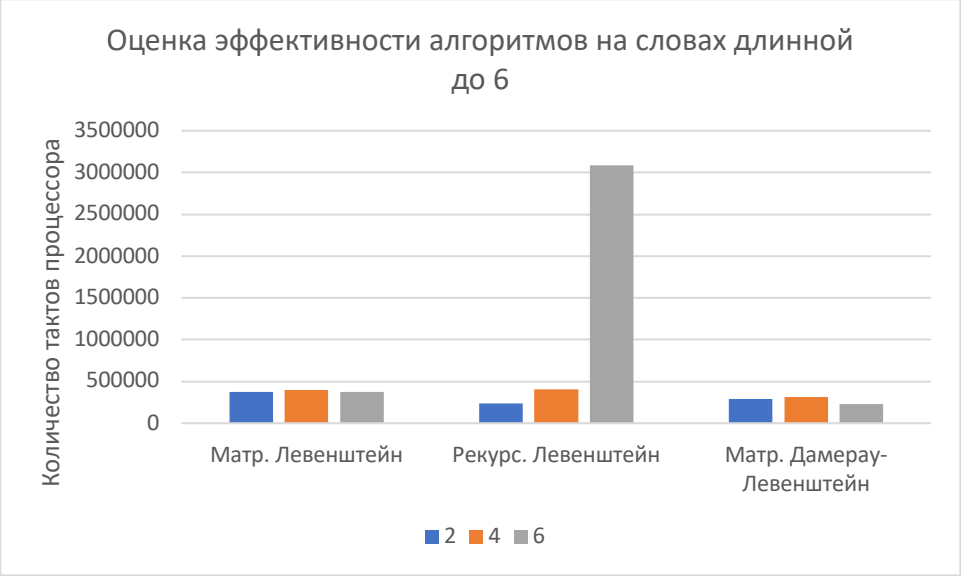

Рисунок 5: Эффективность по времени работы в зависимости от входных слов длиной до 6

Если брать больше 6 слов (см. результаты экспериментов, приведенных на Рис. 5), то диаграмма масштабируется так, что видно только самое большое значение в рекурсивной вариации алгоритма Левенштейна.

Эффективность по памяти для матричных реализаций алгоритмов высчитывается путем определения памяти, выделяемой для матрицы и переменных. Для рекурсивной реализации алгоритма память определяем

путем нахождения количества вызовов алгоритма. Результаты показаны на Таблице 2.1 и Рис. 6.

Таблица 2.1: Эффективность алгоритмов по памяти в байтах

Длина строки	Левенштейн	Дамер.Левеншт.	Дамер.Левеншт.Рекурс.
0	8	12	0
2	24	28	120
4	72	76	551
6	152	156	2615
8	264	268	14232
10	408	412	79384

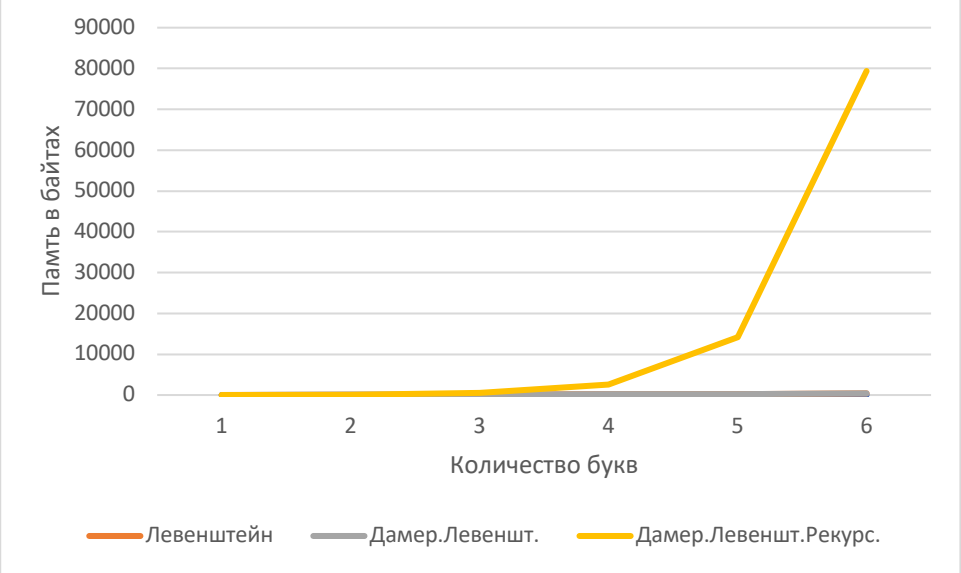

Рис. 6: Эффективность по памяти в зависимости от входных слов длиной до 10

4.3 Сравнительный анализ

По данным эксперимента видно, что при больших входных данных скорость работы рекурсивной реализации алгоритма по нахождению расстояния Левенштейна сильно уменьшается в сравнении с матричной реализацией данного алгоритма. Также при больших входных данных могут возникать сбои программы.

Также алгоритм Дамерау-Левенштейна предпочтительней, в связи с тем, что в данном алгоритме присутствует операция транспозиции. Но если

брать во внимание рекурсивную реализацию алгоритма Дамерау-Левенштейна, то будет наблюдаться такой же экспоненциальный рост времени работы, поэтому матричная реализация более целесообразна.

Таким образом, для проблемы исправления ошибок лучше всего взять во внимание матричную реализацию алгоритма Дамерау-Левенштейна

Заключение

В данной лабораторной работы были исследованы 3 реализации алгоритмов для поиска редакционного расстояния. После проведения некоторого количества экспериментов, был произведен сравнительный анализ, на основе которого была выявлена эффективность матричной реализации двух вариаций алгоритмов.

После проведения экспериментальной части, оказалось, что матричная(итеративная) реализация алгоритмов оказалась намного эффективнее по памяти и по скорости работы, в отличие от рекурсивной. Однако рекурсивная реализация оказалась проще по написанию на языке программирования C++, что было выявлено в технологической части.