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 дисциплине "Анализ алгоритмов"

Тема Редакционные расстояния

Студент Костев Д.И.

Группа ИУ7-51Б

Оценка (баллы) _____

Преподаватели Волкова Л.Л.

Москва — 2021 г.

Оглавление

Введение **3**

1 Аналитическая часть **4**
1.1 Расстояние Левенштейна 4
1.2 Расстояния Дамерау — Левенштейна 5
1.3 Матричный алгоритм нахождения расстояния Левенштейна 6
1.4 Рекурсивный алгоритм нахождения расстояния Левенштейна с заполнением матрицы 6
1.5 Вывод из аналитической части 6

2 Конструкторская часть **7**
2.1 Схемы алгоритмов 7
2.2 Вывод из конструкторской части 11

3 Технологическая часть **12**
3.1 Требование к ПО 12
3.2 Средства реализации 12
3.3 Реализация алгоритмов 12
3.4 Тестовые данные 15
3.5 Вывод из технологической части 15

4 Исследовательская часть **16**
4.1 Технические характеристики 16
4.2 Время выполнения алгоритмов 16
4.3 Использование памяти 17
4.4 Вывод из исследовательской части 17

Заключение **18**

Литература **19**

Введение

Расстояние Левенштейна - минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

Расстояние Левенштейна применяется в теории информации и компьютерной лингвистике для следующих задач:

- исправления ошибок в слове;
- сравнения текстовых файлов утилитой diff;
- в биоинформатике для сравнения генов, хромосом и белков.

Цель лабораторной работы: изучение метода динамического программирования на материале алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

Для достижения поставленной цели требуется решить поставленные задачи.

1. Изучение алгоритмов Левенштейна и Дамерау-Левенштейна.
2. Применение метода динамического программирования для матричной реализации указанных алгоритмов.
3. Получение практических навыков реализации указанных алгоритмов: матричные и рекурсивные версии.
4. Сравнительный анализ линейной и рекурсивной реализаций выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам (времени и памяти).
5. Экспериментальное определение различий во временно́й и емкостной эффективности реализаций алгоритмов.

1 Аналитическая часть

1.1 Расстояние Левенштейна

Расстояние Левенштейна между двумя строками — это минимальное количество операций вставки, удаления и замены, необходимых для превращения одной строки в другую.

Цены операций могут зависеть от вида операции (вставка (insert), удаление (delete), замена (replace)) и/или от участвующих в ней символов, отражая разную вероятность разных ошибок при вводе текста, и т. п. В общем случае:

- $w(a, b)$ — цена замены символа a на символ b;
- $w(\lambda, b)$ — цена вставки символа b;
- $w(a, \lambda)$ — цена удаления символа a.

Для решения задачи о редакционном расстоянии необходимо найти последовательность замен, минимизирующую суммарную цену. Расстояние Левенштейна является частным случаем этой задачи при

- $w(a, a) = 0$,
- $w(a, b) = 1,\ a \neq b$,
- $w(\lambda, b) = 1$,
- $w(a, \lambda) = 1$.

Расстояние Левенштейна между двумя строками a и b может быть вычислено по формуле 1.1, где $|a|$ означает длину строки a; $a[i]$ — i-ый символ строки a , функция $D(i, j)$ определена как:

$$D(i,j) = \begin{cases} 0 & \text{i} = 0, \text{j} = 0 \\ i & \text{j} = 0, \text{i} > 0 \\ j & \text{i} = 0, \text{j} > 0 \\ \min\{ & \\ \quad D(i, j-1) + 1 & \\ \quad D(i-1, j) + 1 & \text{i} > 0, \text{j} > 0 \\ \quad D(i-1, j-1) + m(a[i], b[j]) & (1.2) \\ \} & \end{cases}, \tag{1.1}$$

а функция 1.2 определена как:

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

Рекурсивный алгоритм реализует формулу 1.1. Функция D составлена для решения следующих задач.

1. Для перевода 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 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 проведения любых двух операций можно поменять, порядок проведения операций не имеет никакого значения. Полагая, 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.

1.2 Расстояния Дамерау — Левенштейна

Расстояние Дамерау — Левенштейна может быть найдено по формуле (1.3):

$$d_{a,b}(i,j) = \begin{cases} \max(i,j), & \text{если } \min(i,j) = 0, \\ \min\{ & \\ \quad d_{a,b}(i,j-1) + 1, & \\ \quad d_{a,b}(i-1,j) + 1, & \\ \quad d_{a,b}(i-1,j-1) + m(a[i],b[j]), & \text{иначе} \\ \quad \begin{cases} d_{a,b}(i-2,j-2) + 1, & \text{если } i,j > 1; \\ & a[i] = b[j-1]; \\ & b[j] = a[i-1] \\ \infty, & \text{иначе} \end{cases} & \\ \} & \end{cases}. \tag{1.3}$$

Формула выводится по тем же соображениям, что и формула (1.1). Как и в случае с рекурсивным методом, прямое применение этой формулы неэффективно по времени исполнения, то аналогично методу из 1.4 производится добавление матрицы для хранения промежуточных значений рекурсивной формулы.

1.3 Матричный алгоритм нахождения расстояния Левенштейна

Прямая реализация формулы 1.1 может быть малоэффективна по времени исполнения при больших i, j, т. к. множество промежуточных значений $D(i, j)$ вычисляются заново множество раз подряд. Для оптимизации нахождения расстояния Левенштейна можно использовать матрицу в целях хранения соответствующих промежуточных значений (выполнять меморизацию промежуточных значений). В таком случае алгоритм представляет собой построчное заполнение матрицы $A_{|a|,|b|}$ значениями $D(i, j)$.

1.4 Рекурсивный алгоритм нахождения расстояния Левенштейна с заполнением матрицы

Рекурсивный алгоритм заполнения можно оптимизировать по времени выполнения с использованием матричного алгоритма. Суть данного метода заключается в параллельном заполнении матрицы при выполнении рекурсии. В случае, если рекурсивный алгоритм выполняет расчёт для данных, которые еще не были обработаны (в ячейке матрицы записана машиная бесконечность), результат нахождения расстояния заносится в матрицу. В случае, если обработанные ранее данные встречаются снова (в ячейке матрицы находится значение, отличное от машинной бесконечности), для них расстояние не находится и алгоритм переходит к следующему шагу.

1.5 Вывод из аналитической части

В данном разделе были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна, который является модификацией первого, учитывающего возможность перестановки соседних символов. Формулы Левенштейна и Дамерау — Левенштейна для рассчета расстояния между строками задаются рекурсивно, а следовательно, алгоритмы могут быть реализованы рекурсивно или итерационно.

2 Конструкторская часть

2.1 Схемы алгоритмов

В данной части будут рассмотрены схемы алгоритмов нахождения расстояние Левештейна и Дамерау - Левенштейна. На рисунках 2.1 - 2.4 представлены рассматриваемые алгоритмы.

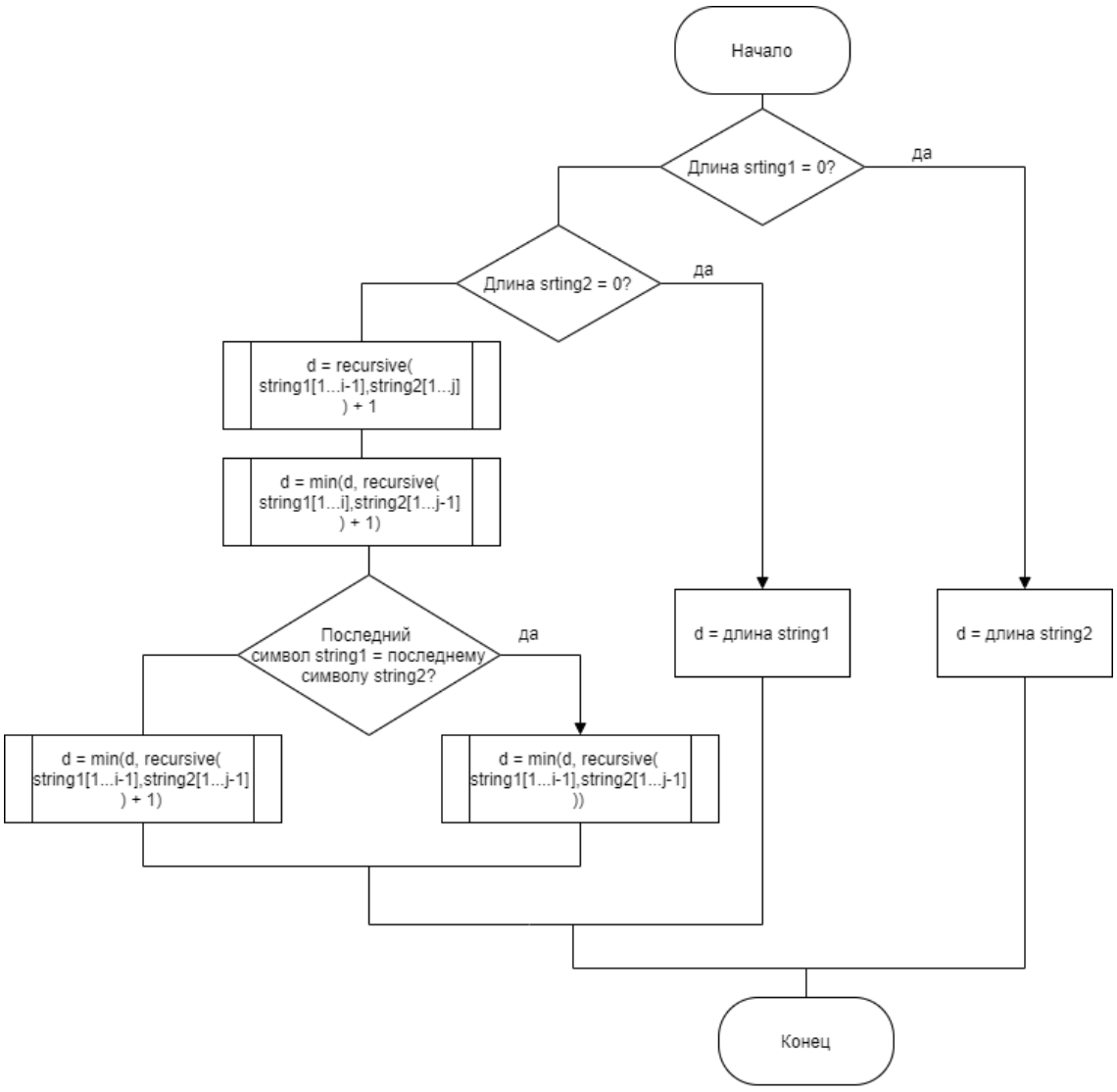

Рис. 2.1: Схема рекурсивного алгоритма нахождения расстояния Левенштейна

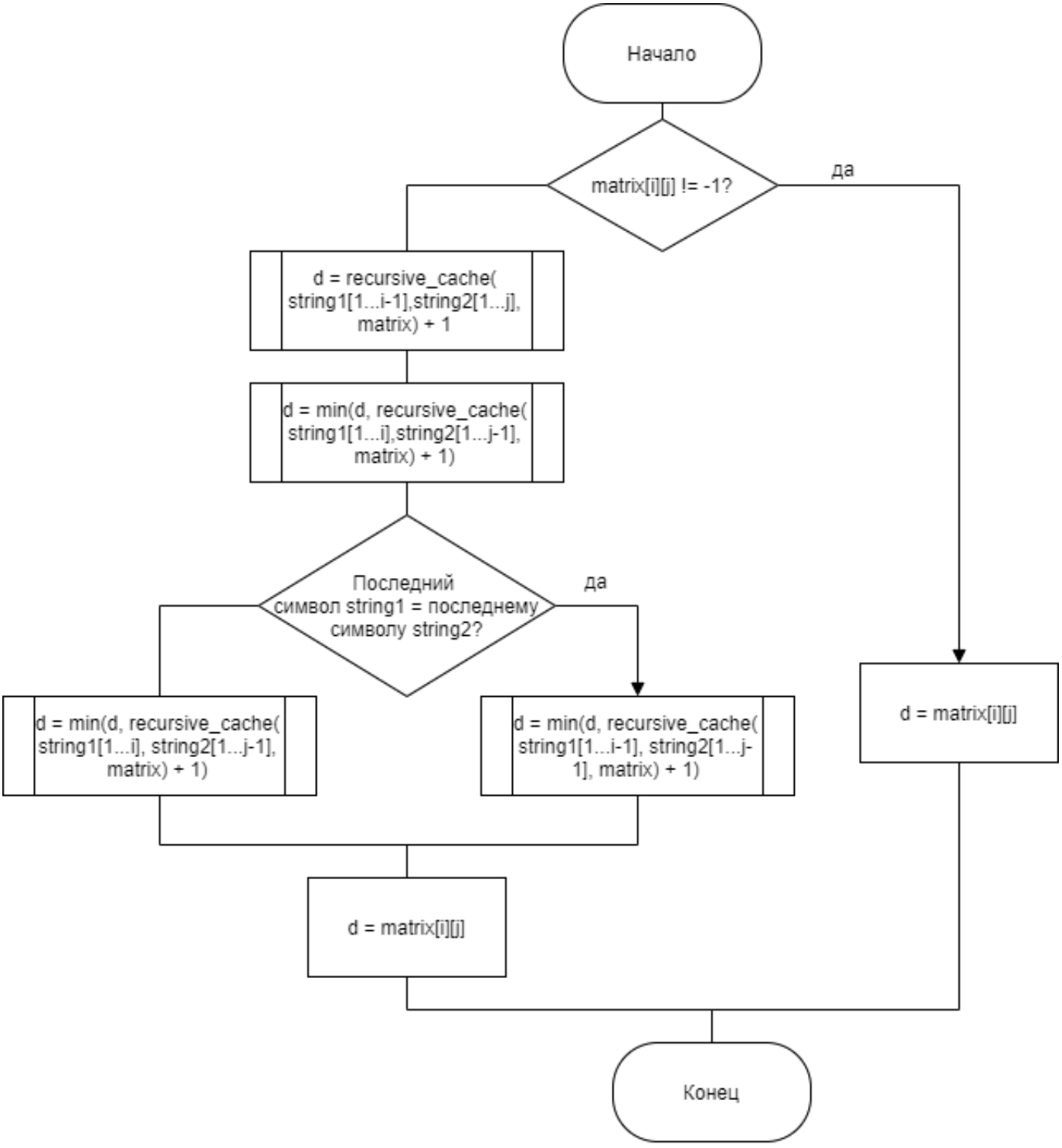

Рис. 2.2: Схема рекурсивного алгоритма с меморизацией значений расстояния Левенштейна

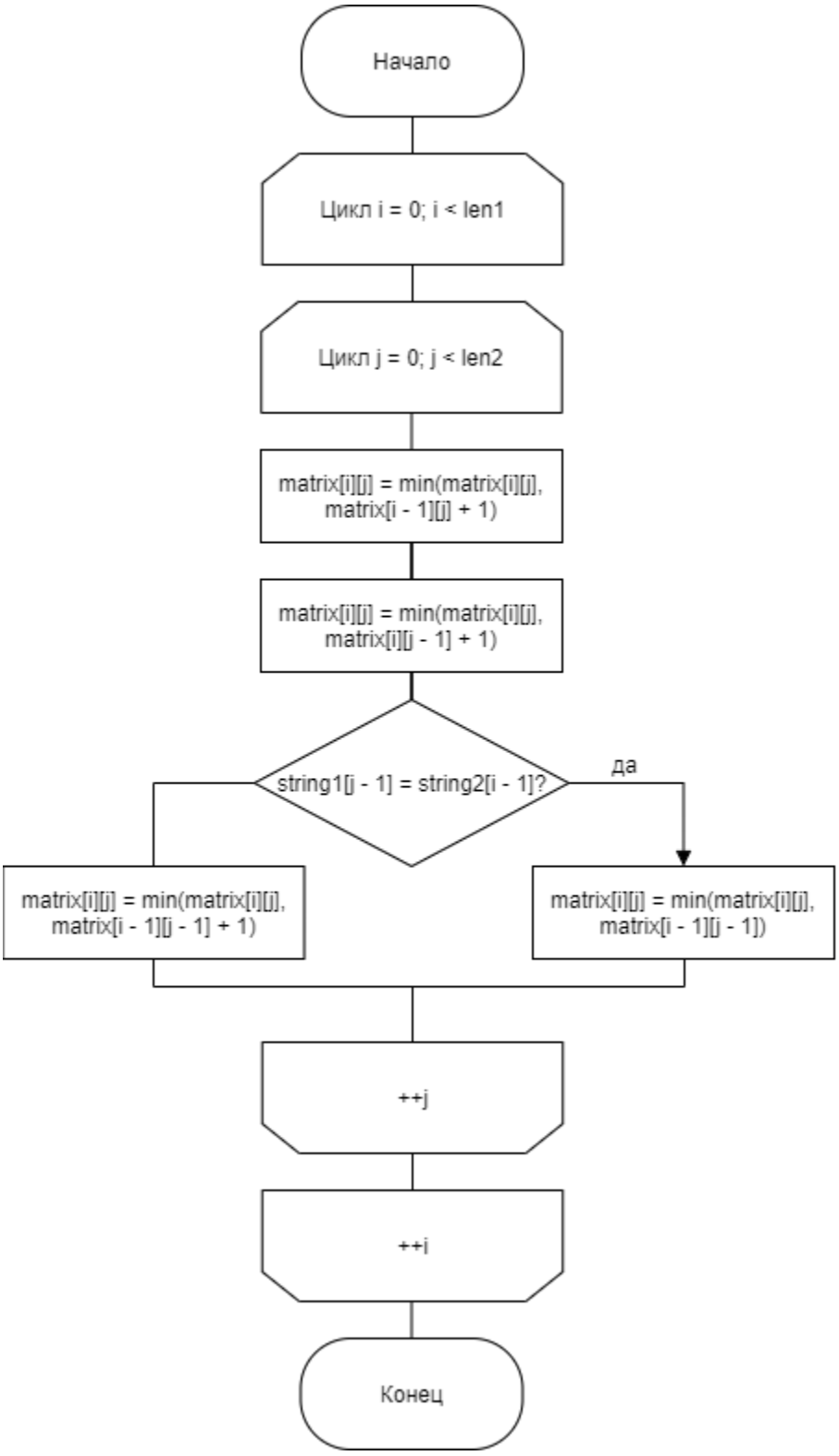

Рис. 2.3: Схема итеративного алгоритма нахождения расстояния Левенштейна

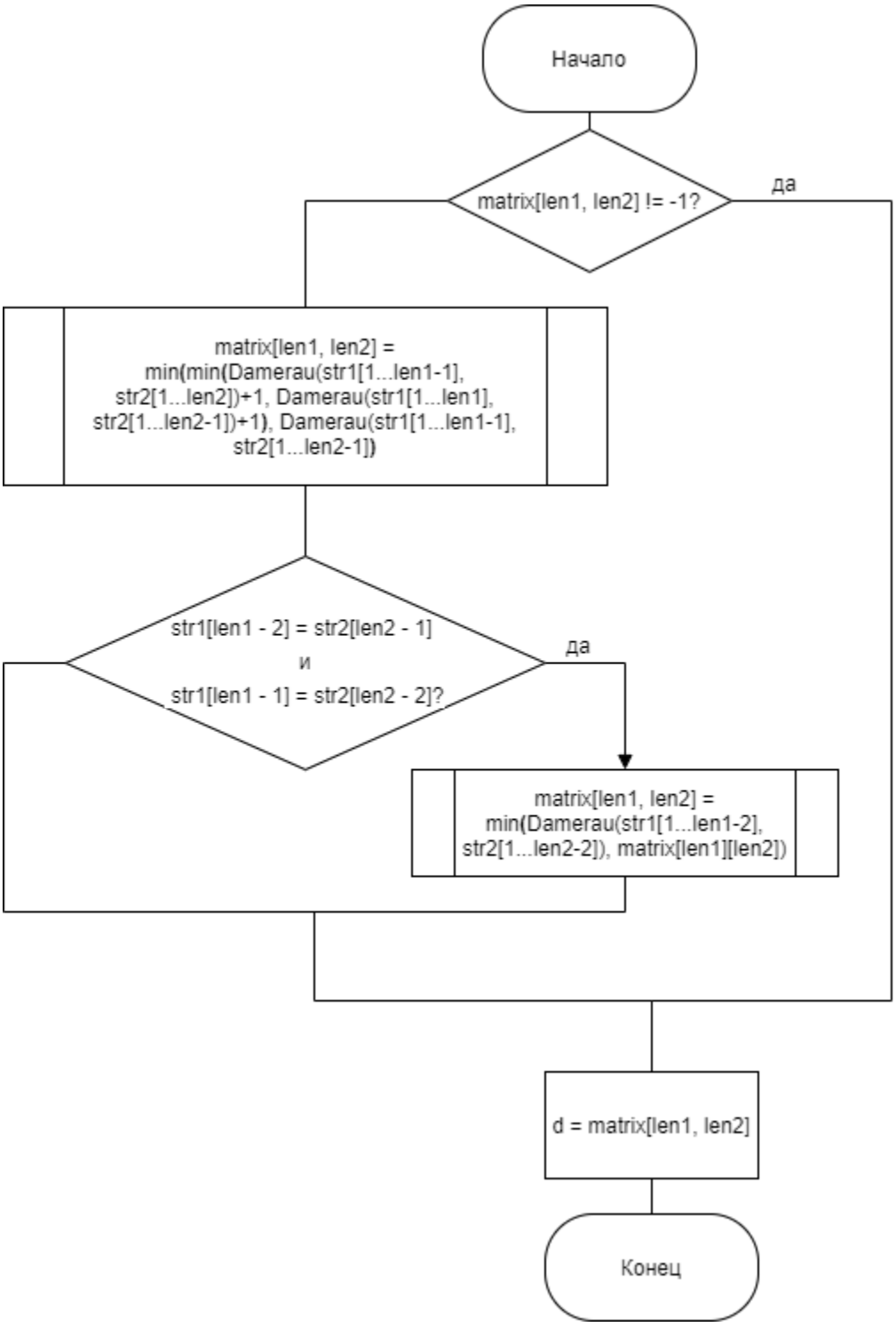

Рис. 2.4: Схема итеративного алгоритма нахождения расстояния Дамерау-Левенштейна

2.2 Вывод из конструкторской части

На основе теоретических данных, полученные в аналитическом разделе, были построены схемы рассматриваемых алгоритмов.

3 | Технологическая часть

3.1 Требование к ПО

Ко вводу предъявляются следующие требования.

1. На вход подаются две строки в любой раскладке (в том числе и пустые).
2. ПО должно выводить полученное расстояние и вспомогательны матрицы.
3. ПО должно выводить потраченную память и время.

3.2 Средства реализации

Для реализации программы нахождения расстояние Левенштейна был выбран язык программирования c++. Для анализа времени использовалась функция getProcessTimes [2].

3.3 Реализация алгоритмов

В листингах 3.1 - 3.4 приведена реализация алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

Листинг 3.1: Функция нахождения расстояния Левенштейна рекурсивно

```
size_t recur(const string &str1, const size_t len1, const string &str2,
    const size_t len2)
{
    if (len1 == len2 && len1 == 0)
        return 0;
    else if (len1 == 0)
        return len2;
    else if (len2 == 0)
        return len1;
    else
    {
        bool flag = str1[len1 - 1] != str2[len2 - 1];
        return min(min(recur(str1, len1 - 1, str2, len2) + 1,
                       recur(str1, len1, str2, len2 - 1) + 1),
                   recur(str1, len1 - 1, str2, len2 - 1) + flag);
    }
}
```

Листинг 3.2: Функция нахождения расстояние Левенштейна рекурсивно с меморизацией

```
size_t rec_cache(const string &str1, const size_t len1, const string &str2,
    const size_t len2, size_t **matr)
{
    if (matr[len1][len2] != 0)
        return matr[len1][len2];
    else if (len1 == len2 && len1 == 0)
        matr[len1][len2] = 0;
    else if (len1 == 0)
        matr[len1][len2] = len2;
    else if (len2 == 0)
        matr[len1][len2] = len1;
    else
    {
        bool flag = str1[len1 - 1] != str2[len2 - 1];
        matr[len1][len2] = min(min(rec_cache(str1, len1 - 1, str2, len2,
            matr) + 1,
                                   rec_cache(str1, len1, str2, len2 - 1,
                                       matr) + 1),
                               rec_cache(str1, len1 - 1, str2, len2 - 1,
                                   matr) + flag);
    }
    return matr[len1][len2];
}

size_t rec_cache_method(const string &str1, const string &str2)
{
    size_t len1 = str1.length(), len2 = str2.length();
    auto matr = new size_t *[len1 + 1];
    for (size_t i = 0; i < len1 + 1; i++)
        matr[i] = new size_t[len2 + 1];
    for (size_t i = 0; i < len1 + 1; i++)
        for (size_t j = 0; j < len2 + 1; j++)
            matr[i][j] = 0;

    rec_cache(str1, len1, str2, len2, matr);
    size_t result = matr[len1][len2];
    for (size_t i = 0; i < len1 + 1; i++)
        delete matr[i];
    delete matr;
    return result;
}
```

Листинг 3.3: Функция нахождения расстояния Левенштейна итеративно

```
size_t matr_method(const string &str1, const string &str2)
{
    size_t n = str1.length() + 1, m = str2.length() + 1;
    auto matr = new size_t *[n];
    for (size_t i = 0; i < n; i++)
```

```
    {
        matr[i] = new size_t[m];
        matr[i][0] = i;
    }
    for (size_t i = 0; i < m; i++)
        matr[0][i] = i;

    for (size_t i = 1; i < n; i++)
        for (size_t j = 1; j < m; j++)
            matr[i][j] = min(min(matr[i - 1][j] + 1, matr[i][j - 1] + 1),
                matr[i - 1][j - 1] + (str1[i - 1] == str2[j - 1] ? 0 : 1));

    size_t result = matr[n - 1][m - 1];
    for (size_t i = 0; i < n; i++)
        delete matr[i];
    delete matr;
    return result;
}
```

Листинг 3.4: Функция нахождения расстояния Дамерау-Левенштейна матрично

```
size_t Damerau(const string &str1, const size_t len1, const string &str2,
    const size_t len2, size_t **matr)
{
    if (matr[len1][len2] != 0)
        return matr[len1][len2];
    else if (len1 == len2 && len1 == 0)
        matr[len1][len2] = 0;
    else if (len1 == 0)
        matr[len1][len2] = len2;
    else if (len2 == 0)
        matr[len1][len2] = len1;
    else
    {
        bool flag = str1[len1 - 1] != str2[len2 - 1];
        matr[len1][len2] = min(min(Damerau(str1, len1 - 1, str2, len2, matr)
            + 1,
                                   Damerau(str1, len1, str2, len2 - 1, matr)
                                       + 1),
                               Damerau(str1, len1 - 1, str2, len2 - 1, matr)
                                   + flag);

        if (len1 > 1 && len2 > 1 && str1[len1 - 2] == str2[len2 - 1] && str1
            [len1 - 1] == str2[len2 - 2])
            matr[len1][len2] = min(Damerau(str1, len1 - 2, str2, len2 - 2,
                matr) + 1, matr[len1][len2]);
    }
    return matr[len1][len2];
}

```

```
size_t Damerau_method(const string &str1, const string &str2)
{
    size_t len1 = str1.length(), len2 = str2.length();
    auto matr = new size_t *[len1 + 1];
    for (size_t i = 0; i < len1 + 1; i++)
        matr[i] = new size_t[len2 + 1];
    for (size_t i = 0; i < len1 + 1; i++)
        for (size_t j = 0; j < len2 + 1; j++)
            matr[i][j] = 0;

    Damerau(str1, len1, str2, len2, matr);

    size_t result = matr[len1][len2];
    for (size_t i = 0; i < len1 + 1; i++)
        delete matr[i];
    delete matr;
    return result;
}
```

3.4 Тестовые данные

В таблице 3.1 приведены тестовые данные, на которых было протестированно разработанное ПО. Тестирование проводилось по методу черного ящика. Все тесты пройдены успешно.

Таблица 3.1: Тестовые данные

№	Первое слово	Второе слово	Ожидаемый результат	Полученный результат
1			0 0 0 0	0 0 0 0
2	kot	skat	2 2 2 2	2 2 2 2
3	kate	ktae	2 2 1 1	2 2 1 1
4	abacaba	aabcaab	4 4 2 2	4 4 2 2
5	sobaka	sboku	3 3 3 3	3 3 3 3
6	qwerty	queue	4 4 4 4	4 4 4 4
7	aaaa	bbbbb	5 5 5 5	5 5 5 5
8		abc	3 3 3 3	3 3 3 3
9	parallels		9 9 9 9	9 9 9 9
10	буквы	букыв	2 2 2 1	2 2 2 1

3.5 Вывод из технологической части

В данном разделе были разработаны исходные коды четырех алгоритмов: вычисления расстояния Левенштейна рекурсивно, с заполнением матрицы и рекурсивно с заполнением матрицы, а также вычисления расстояния Дамерау — Левенштейна с заполнением матрицы.

4 | Исследовательская часть

4.1 Технические характеристики

Ниже приведеные технические характеристики устройства, на котором было проведенно тестирование ПО.

- Операционная система: Windows 10 64-bit.
- Оперативная память: 8 ГБ.
- Процессор: Intel(R) Core(TM) i5-8250 CPU @ 1.60GHz.

4.2 Время выполнения алгоритмов

Время выполнения алгоритм замерялось с помощью применения технологии профайлинга. Данный инстрмуент даёт детальное описание количества вызовов и количества времени CPU, занятого каждой функцией.

В таблице 4.1. представленны замеры времени работы для каждого из алгоритмов. Рекурсивный алгоритм проводился на строках размером до 10 символов, из-за слишком быстрого увеличения времени работы программы с увеличением длины строки.

Таблица 4.1: Таблица времени выполнения алгоритмов (в милисекундах)

Длина строки	Итеративный	Рекурсивный	Рек. с меморизацией	Дамерау-Левенштейн
5	0.00078125	0.03125	0.00109375	0.00125
6	0.0009375	0.078125	0.0015625	0.00171875
7	0.00125	0.4375	0.00203125	0.00203125
8	0.0015625	2.3125	0.0028125	0.0028125
9	0.0021875	13.625	0.00328125	0.0034375
10	0.003125	–	0.0046875	0.0046875
20	0.0078125	–	0.0171875	0.015625
30	0.0171875	–	0.028125	0.0328125
40	0.0234375	–	0.046875	0.05625
50	0.0421875	–	0.078125	0.084375

4.3 Использование памяти

Алгоритмы нахождения расстояния Левенштейна и Дамерау — Левенштейна не отличаются друг от друга с точки зрения использования памяти.

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

$$(\mathbb{S}(STR_1) + \mathbb{S}(STR_2)) \cdot (2 \cdot \mathbb{S}(\text{string}) + 3 \cdot \mathbb{S}(\text{integer})), \tag{4.1}$$

где $\mathbb{S}$ — оператор вычисления размера, STR_1, STR_2 — строки, string — строковый тип, integer — целочисленный тип.

Использование памяти при итеративной реализации теоретически равно:

$$(\mathbb{S}(STR_1) + 1) \cdot (\mathbb{S}(STR_2) + 1) \cdot \mathbb{S}(\text{integer}) + 5 \cdot \mathbb{S}(\text{integer}) + 2 \cdot \mathbb{S}(\text{string}). \tag{4.2}$$

4.4 Вывод из исследовательской части

Рекурсивная реализация алгоритма нахождения расстояния Левенштейна работает дольше итеративной реализации, время работы этой реализации увеличивается в геометрической прогрессии. Рекурсивный с матрицей и Дамерау-Левенштейна работают примерно за одно время и лишь немного уступают итеративному алгоритму.

Заключение

В ходе проделанной работы были выполнены следующие задачи.

1. Был изучен метод динамического программирования на материале реализации алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

2. Были изучены алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками.

3. Получены практические навыки раелизации указанных алгоритмов в матричной и рекурсивных версиях, а так же в версиях с мемоизацией.

4. Был произведен сравнительный анализ линейной и рекурсивной реализаций выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам (времени и памяти).

5. Экспериментально было подтверждено различие во временной эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработаного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк.

Рекурсивная реализация алгоритма Левенштейна проигрывает нерекурсивной по времени исполнения в несколько десятков раз. Так же стоит отметить, что итеративный алгоритм Левештейна выполняется немного быстрее, чем итеративный алгоритм Дамерау - Левенштейна, но в целом алгоритмы выполняются за примерно одинаковое время.

Теоретически было рассчитано использования памяти в каждой из реализаций алгоритмов нахождения расстояния Левенштейна и Дамерау - Левенштейна. Алгоритм итеративный использует несколько большее количество памяти.

Из этого можно сделать вывод: если требуется самый быстрое решение задачи, то нужно использовать итеративный алгоритм, если же нужно сберечь память - Дамерау-Левенштейна.

Литература

[1] Кормен Т. Алгоритмы: построение и анализ [Текст] / Кормен Т. - Вильямс, 2014. - 198 с. - 219 с.

[2] GetProcessTimes function. URL: https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-getprocesstimes, 01.10.2021

[3] Левенштейн В.И. Двоичные коды с исправлением выпадений, вставок и замещением символов. Доклады АН СССР 1965. Т. 163. С. 845-848.