**Государственное образовательное учреждение высшего профессионального образования
“Московский государственный технический университет имени Н.Э.Баумана”**

Дисциплина: Анализ алгоритмов

Лабораторная работа №1

Расстояние Левенштейна

Студент группы ИУ7-55Б,
Руднев К. К.,

Преподаватель,
Волкова Л. Л.,
Строганов Ю. В.

2019 г.

Введение

Цель работы: изучение метода динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна.
Задачи работы:

1. изучение алгоритмов Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;
2. применение метода динамического программирования для матричной реализации указанных алгоритмов;
3. получение практических навыков реализации указанных алгоритмов: двух алгоритмов в матричной версии и одного из алгоритмов в рекурсивной версии;
4. сравнительный анализ линейной и рекурсивной реализаций выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);
5. экспериментальное подтверждение различий во временно́й эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк;
6. описание и обоснование полученных результатов в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

1 Аналитическая часть

В рамках раздела будет дано аналитическое описание алгоритмов Левенштейна и Дамерау-Левенштейна.

1.1 Описание алгоритмов

Расстояние Левенштейна (также известно как редакционное расстояние) между двумя строками есть минимальное число операций удаления, вставки, замены символа строки для преобразования одной строки к другой. Для решения поставленной задачи существует несколько возможных алгоритмов. Среди них алгоритмы Левенштейна и Дамерау-Левенштейна.

Допустим, имеются две строки S1 и S2, их длины соответсвенно равны M и N. Тогда для для нахождения расстояния Левенштейна можно воспользоваться следующей формулой:

$$D(i,j) = \begin{cases} max(i,j)\ if\ min(i,j) == 0 \\ min \begin{cases} D(i,j-1)+1 \\ D(i-1,j)+1 \\ D(i-1,j-1)+(S1[i] <> S2[j]) \end{cases} \end{cases}$$

Для нахождения расстояния Дамерау-Левенштейна можно воспользоваться следующей формулой:

$$D(i,j) = \begin{cases} max(i,j)\ if\ min(i,j) == 0 \\ min \begin{cases} D(i,j-1)+1 \\ D(i-1,j)+1 \\ D(i-1,j-1)+(S1[i] <> S2[j]) \\ D(i-2,j-2)+1 \end{cases} & if\ i,j > 1\ and\ Transpose \\ min \begin{cases} D(i,j-1)+1 \\ D(i-1,j)+1 \\ D(i-1,j-1)+(S1[i] <> S2[j]), \end{cases} \end{cases}$$

где $Transpose = S1[i] == S2[j-1]\ and\ S1[i-1] == S2[j]$

2 Конструкторская часть

В дальнейшем на рисунках 1-8 будут представлены схемы расматриваемых алгоритмов. Рисунки 1-2 изображают схему рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна. Рисунки 3-5 изображают схему матричного алгоритма нахождения расстояния Дамерау-Левенштейна. Рисунки 6-8 изображают схему матричного алгоритма нахождения расстояния Левенштейна.

2.1 Разработка алгоритмов

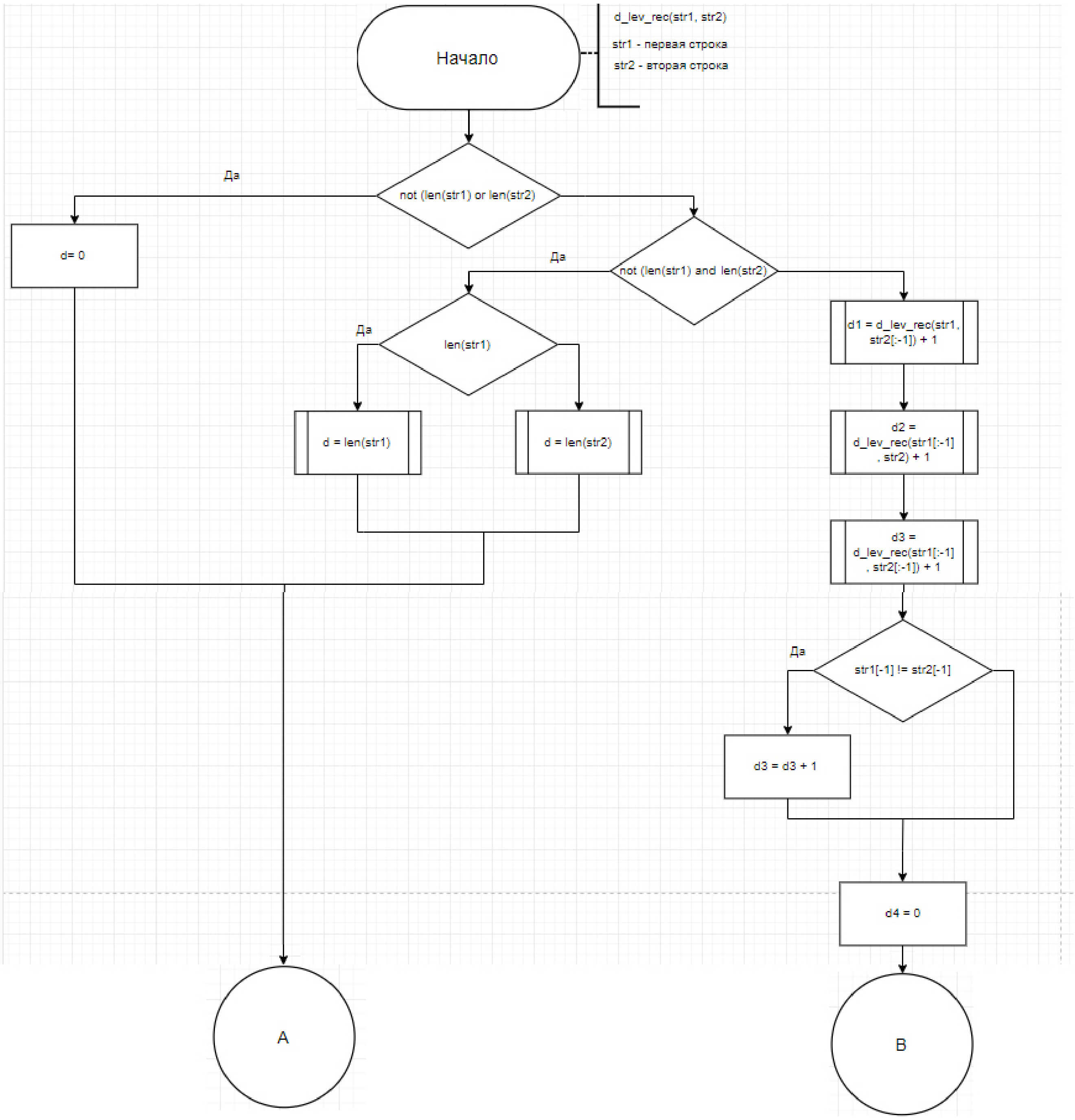


Рис. 1: Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна. Часть 1

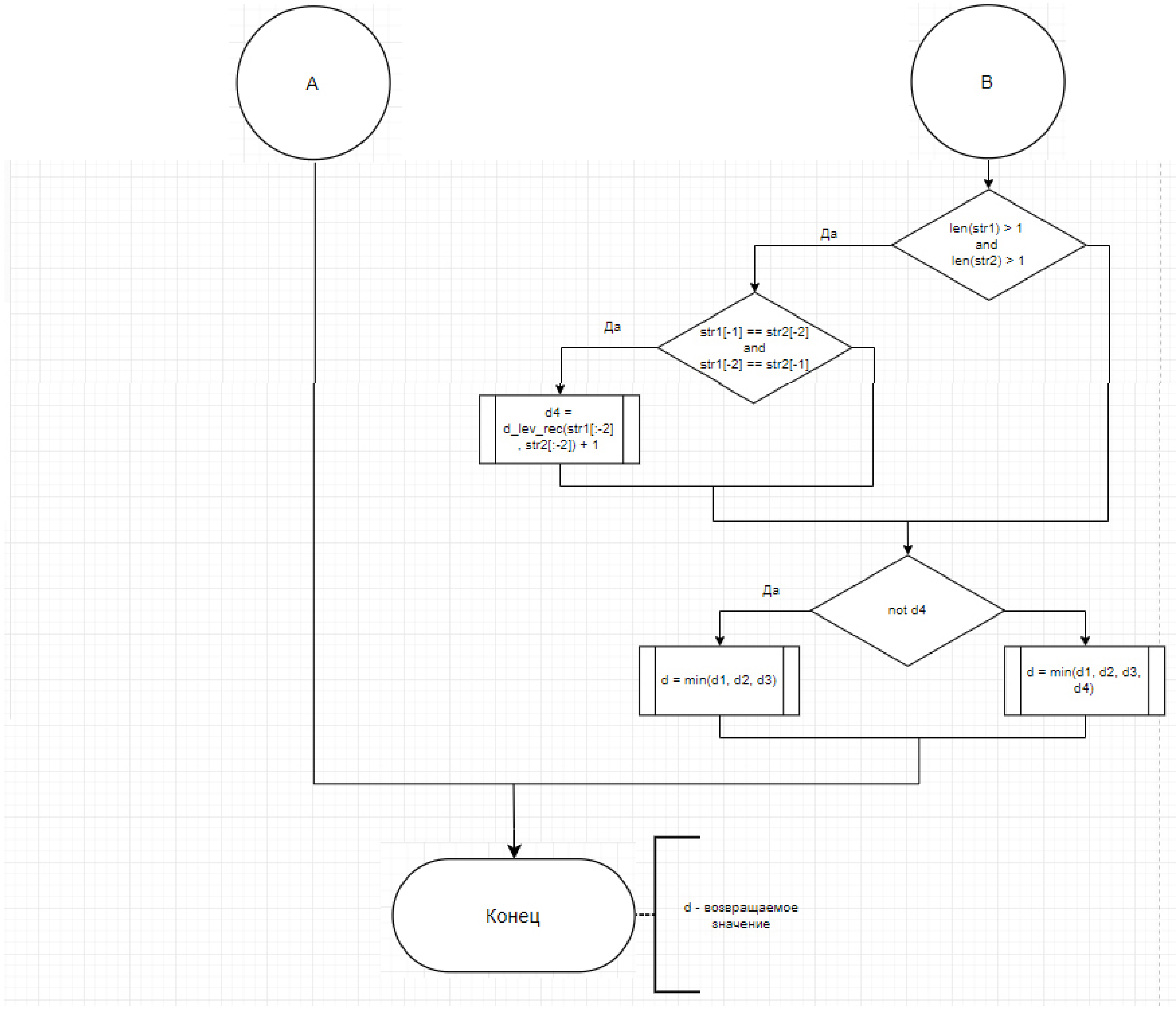


Рис. 2: Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна. Часть 2

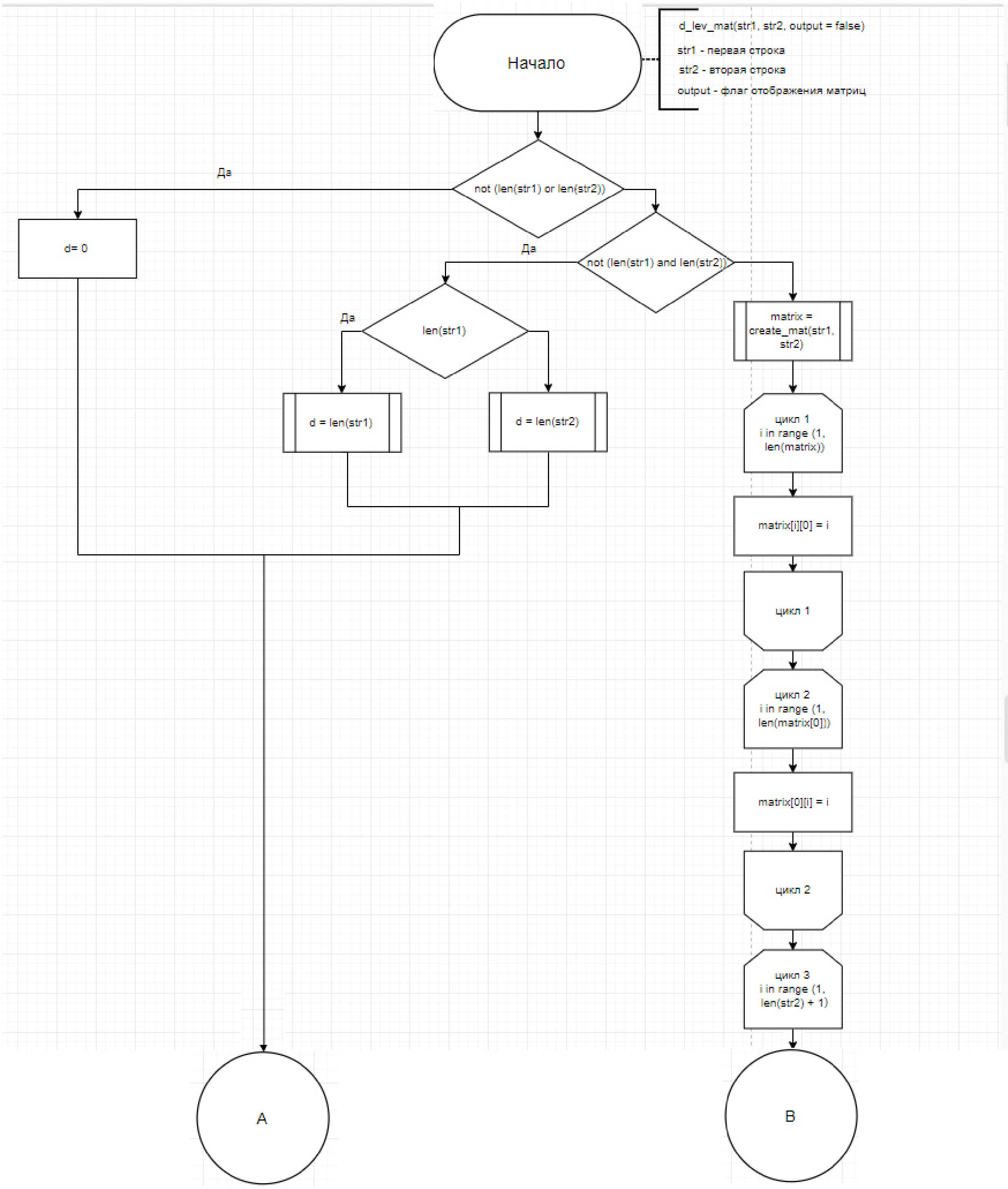


Рис. 3: Матричный алгоритм нахождения расстояния Дамерау-Левенштейна. Часть 1

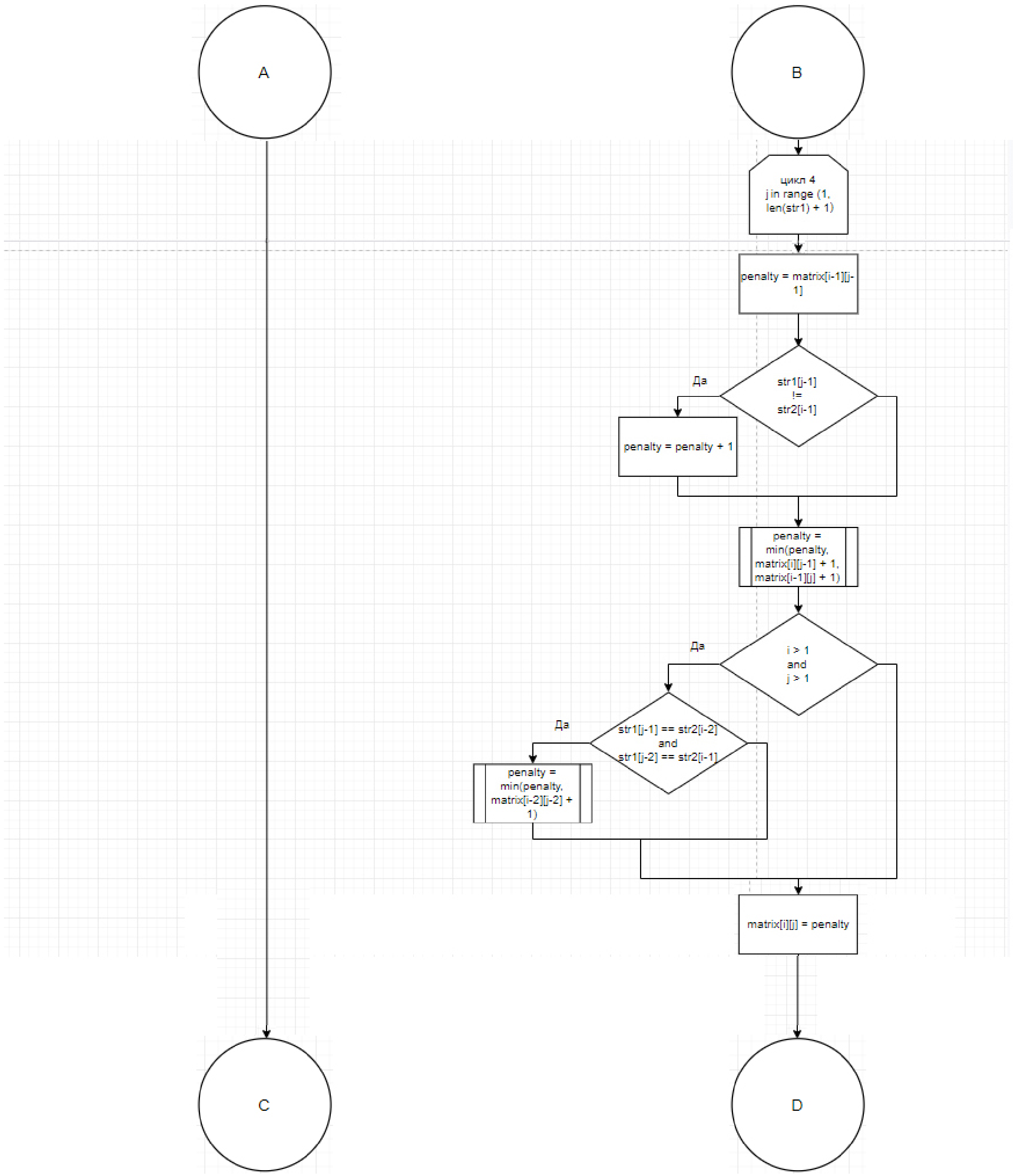


Рис. 4: Матричный алгоритм нахождения расстояния Дамерау-Левенштейна. Часть 2

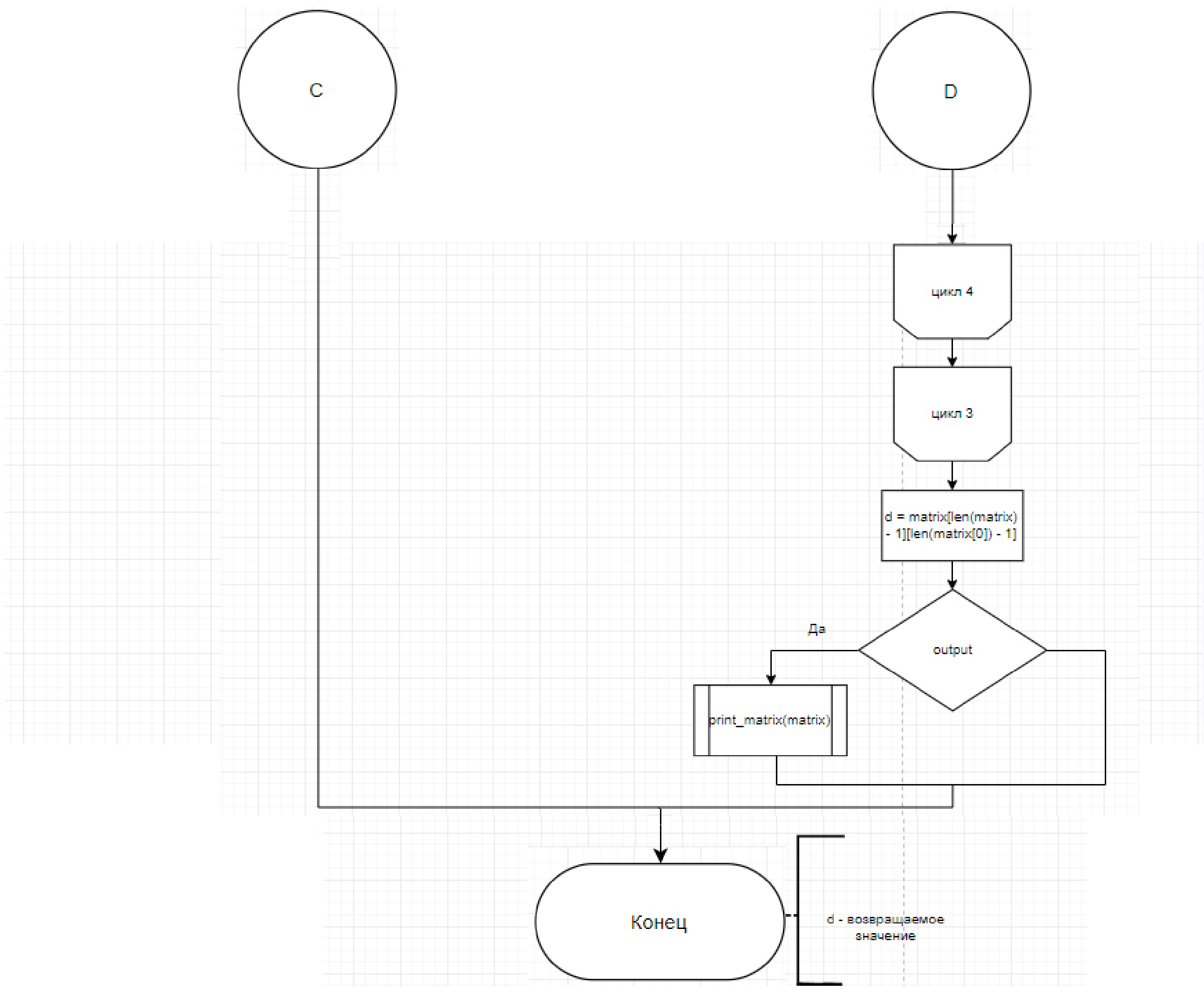


Рис. 5: Матричный алгоритм нахождения расстояния Дамерау-Левенштейна. Часть 3

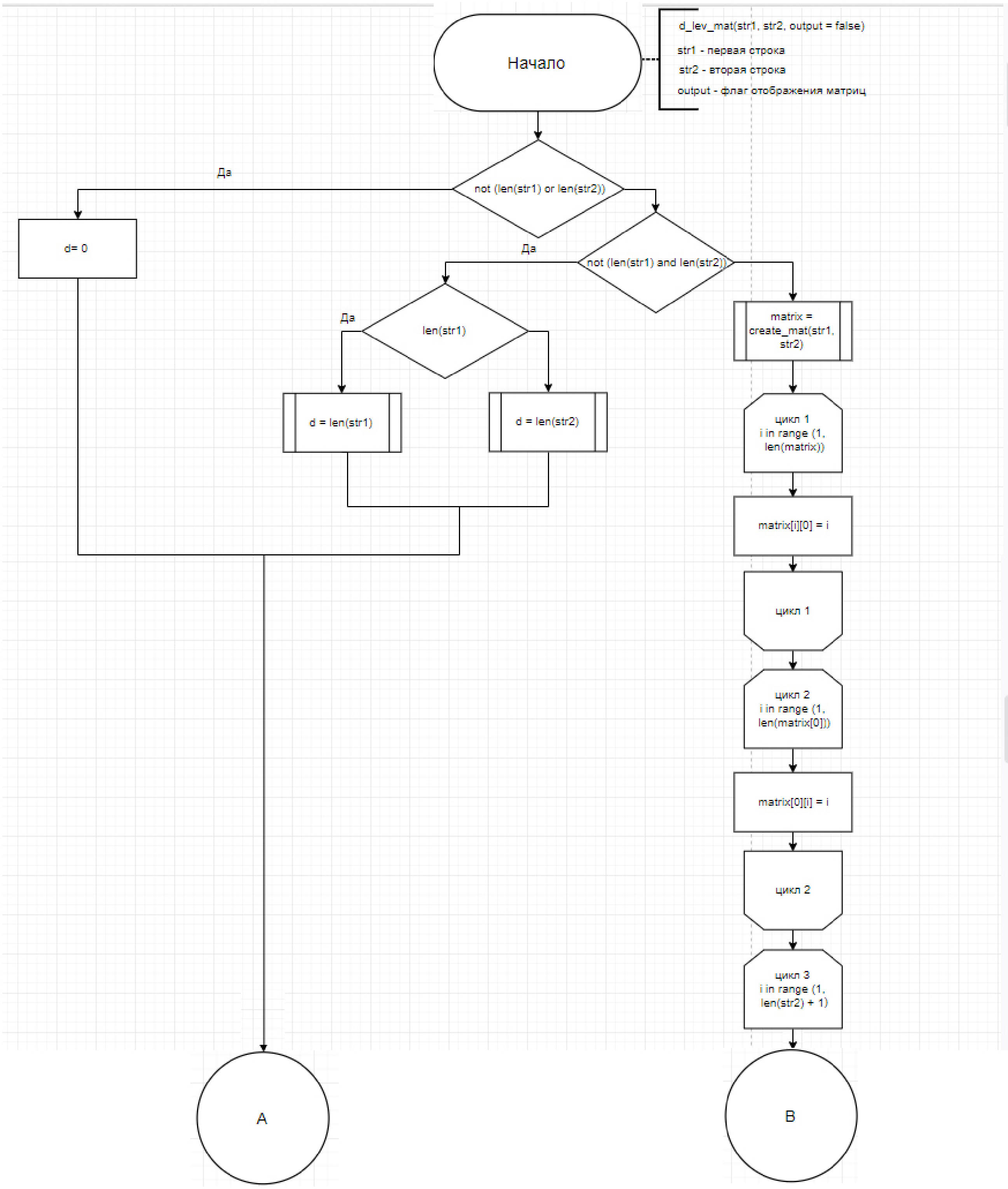


Рис. 6: Матричный алгоритм нахождения расстояния Левенштейна. Часть 1

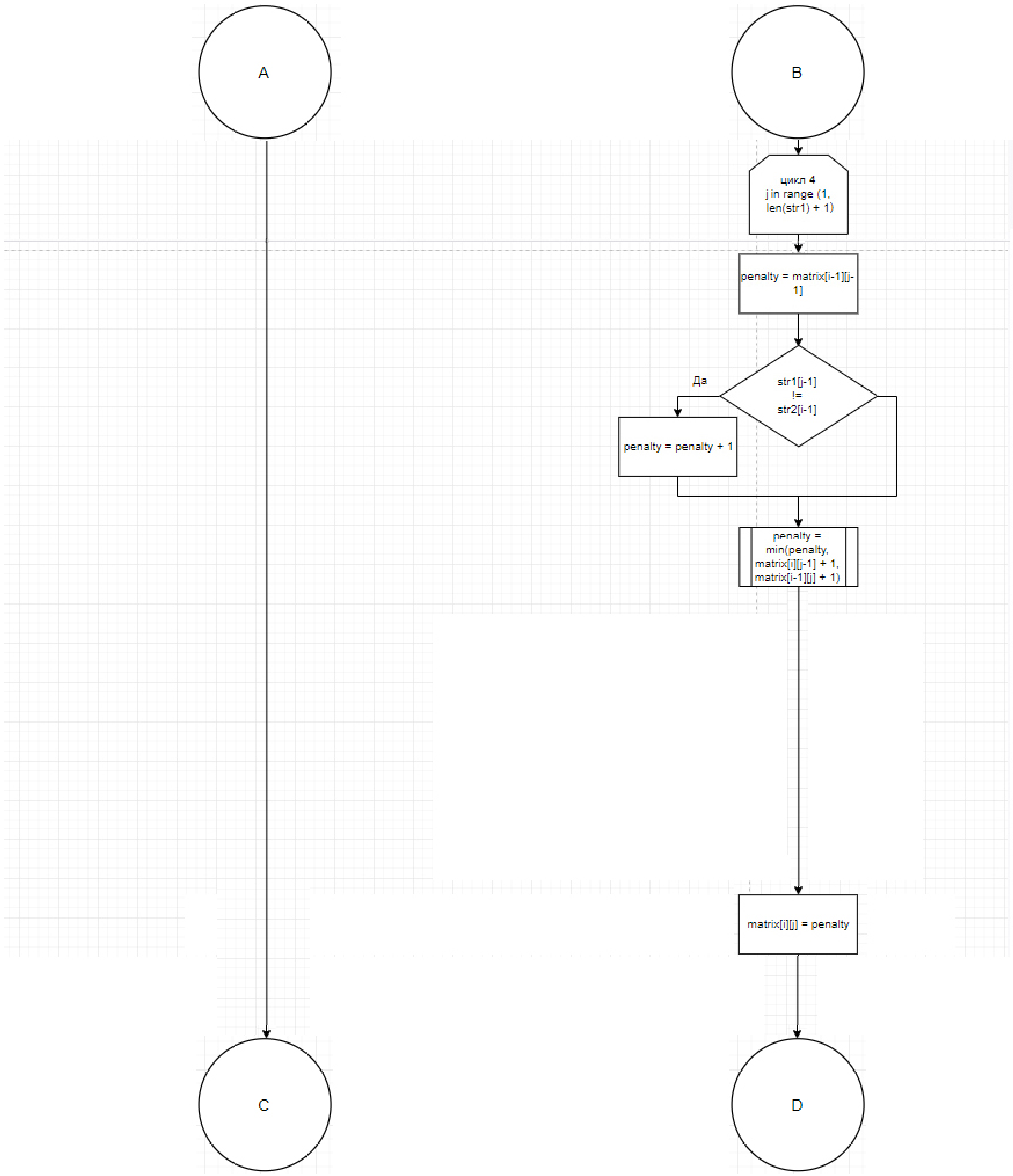


Рис. 7: Матричный алгоритм нахождения расстояния Левенштейна. Часть 2

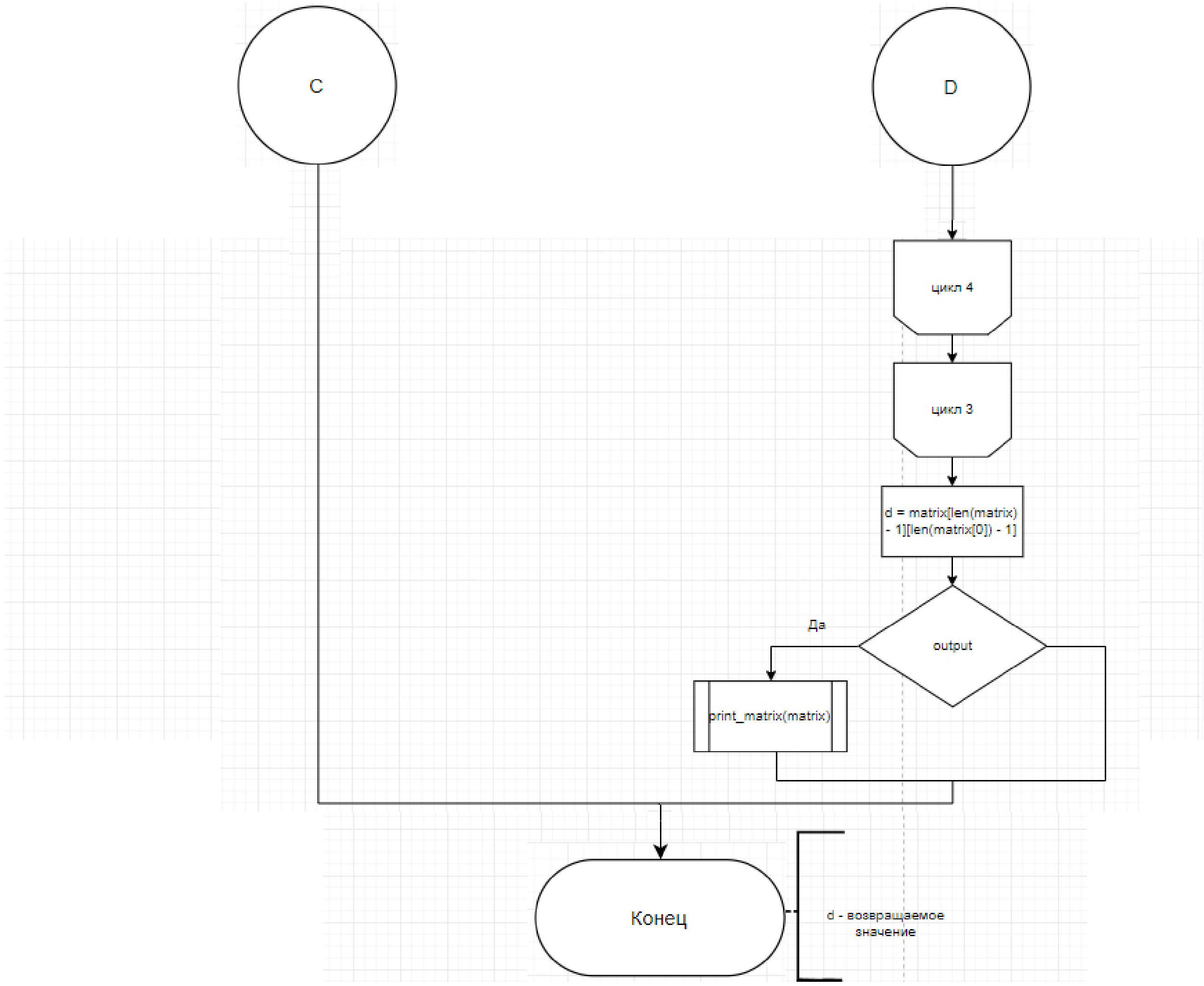


Рис. 8: Матричный алгоритм нахождения расстояния Левенштейна. Часть 3

3 Технологическая часть

В рамках раздела будут описаны инструментарии разработки, выбор среды, требобования к ПО. Также будут предоставлены листинги конкретных реализаций алгоритмов.

3.1. Средства реализации

Для реализации алгоритмов использовался язык программирования Python 3.6.0 и среда разработки PyCharm Community Edition 2017.2.4 by JetBrains. У меня есть определенный опыт работы с данным языком, которого будет достаточно для реализации текущей лабораторной работы, а среда разработки имеет бесплатную комьюнити версию и удобный интерфейс, упрощающий разработку приложения/скрипта.
Замер времени реализован с помощью функции process_time() библиотеки time.
Измеряется время исполнения кода чистого алгоритма (без учета времени на создание матриц, генерацию данных и т.п.).
Замер памяти реализован с помощью функции getsizeof() библиотеки sys. Измеряется максимальное значение памяти, используемой для работы алгоритма.

3.2. Требования к программному обеспечению

На вход программа должна получать две строки, между которыми вычисляется расстояние Левенштейна и/или Дамерау-Левенштейна. На выход программа должна выдавать найденное расстояние Левенштейна и/или Дамерау-Левенштейна, а также по условиям лабораторной работы, в случае использования матричного алгоритма, расчетную матрицу.

3.3. Листинг кода

Листинг 1: Листинг вспомогательных функций и объявлений

```python
import random, time, sys

def print_matrix(mat):
    print("")
    for i in range (len(mat)):
        print(mat[i])

def create_matrix(weight, height):
    res = []
    for i in range (len(height) + 1):
        res.append([0]*(len(weight) + 1))

    return res
```

Листинг 2: Листинг матричного алгоритма расстояния Левенштейна

```python
def lev_mat(str1, str2, output = False):
    if len(str1) and len(str2):
        matrix = create_matrix(str1, str2)

        for i in range (1, len(matrix)):
            matrix[i][0] = matrix[i-1][0] + 1

        for i in range (1, len(matrix[0])):
            matrix[0][i] = matrix[0][i-1] + 1
```

```
        for i in range (1, len(str2) + 1):
            for j in range (1, len(str1) + 1):
                penalty = matrix[i-1][j-1]
                if str1[j-1] != str2[i-1]:
                    penalty += 1
                penalty = min(penalty, matrix[i][j-1] + 1,\
                                        matrix[i-1][j] + 1)

                matrix[i][j] = penalty

        d = matrix[len(matrix) - 1][len(matrix[0]) - 1]
        if output:
            print_matrix(matrix)
    elif not len(str1) and not len(str2):
        d = 0
    else:
        if len(str1):
            d = len(str1)
        else:
            d = len(str2)

    return d
```

Листинг 3: Листинг рекурсивного алгоритма расстояния Дамерау-Левенштейна

```
def d_lev_rec(str1, str2):
    if not (len(str1) or len(str2)):
        return 0
    elif not (len(str1) and len(str2)):
        if len(str1):
            return len(str1)
        else:
            return len(str2)

    d1 = d_lev_rec(str1, str2[:-1]) + 1
    d2 = d_lev_rec(str1[:-1], str2) + 1
    d3 = d_lev_rec(str1[:-1], str2[:-1])
    if str1[-1] != str2[-1]:
        d3 += 1

    d4 = 0
    if len(str1) > 1 and len(str2) > 1:
        if str1[-1] == str2[-2] and str1[-2] == str2[-1]:
            d4 = d_lev_rec(str1[:-2], str2[:-2]) + 1

    if not d4:
        res = min(d1, d2, d3)
    else:
        res = min(d1, d2, d3, d4)

    return res
```

Листинг 4: Листинг матричного алгоритма расстояния Дамерау-Левенштейна

```
def d_lev_mat(str1, str2, output = False):
    if len(str1) and len(str2):
        matrix = create_matrix(str1, str2)

        for i in range (1, len(matrix)):
            matrix[i][0] = matrix[i-1][0] + 1

        for i in range (1, len(matrix[0])):
            matrix[0][i] = matrix[0][i-1] + 1

        for i in range (1, len(str2) + 1):
            for j in range (1, len(str1) + 1):
                penalty = matrix[i-1][j-1]
                if str1[j-1] != str2[i-1]:
                    penalty += 1
                penalty = min(penalty, matrix[i][j-1] + 1,\
                                    matrix[i-1][j] + 1)
                if i > 1 and j > 1:
                    if str1[j-1] == str2[i-2] and\
                                    str2[i-1] == str1[j
                                        -2]:
                        penalty = min(penalty,\
                                    matrix[i-2][j-2] + 1)

                matrix[i][j] = penalty

        d = matrix[len(matrix) - 1][len(matrix[0]) - 1]
        if output:
            print_matrix(matrix)
    elif not len(str1) and not len(str2):
        d = 0
    else:
        if len(str1):
            d = len(str1)
        else:
            d = len(str2)

    return d
```

3.3 Описание возможной занимаемой памяти

Для матричного алгоритма занимаемую память всегда можно оценить формулой (1):

$$mem = (len(str1) + 1) * (len(str2) + 1) + sizeof(int) \quad (1)$$

Для рекурсивного алгоритма Дамерау-Левенштейна можно выразить формулой (2):

$$
\begin{aligned}
mem = \sum_{i=1}^{len(str1)+len(str2)} & 3 * sizeof(str1_i + str2_i) - 4 \\
&+ \sum_{i=3}^{len(str1)+len(str2)} 4 * sizeof(str1_i + str2_i) - 8 \\
&= \sum_{i=1}^{len(str1)+len(str2)} 3 * (50 + len(str1_i + len(str2_i))) \\
&\quad + \sum_{i=3}^{len(str1)+len(str2)} 4 * (50 + len(str1_i + len(str2_i))) - 12 \qquad (2)
\end{aligned}
$$

4 Экспериментальная часть

В рамках раздела будут предоставлены тесты программы, представленные на рисунках 9-14. Будут проведены эксперименты по вычислению времени выполнения алгоритмов, а также занимаемой памяти.

4.1. Примеры работы

```
first string to compare: aa
second string to compare: aaaa
string1: ( aa )  2
string2: ( aaaa )  4

[0, 1, 2]
[1, 0, 1]
[2, 1, 0]
[3, 2, 1]
[4, 3, 2]
levenstein matrix  2

[0, 1, 2]
[1, 0, 1]
[2, 1, 0]
[3, 2, 1]
[4, 3, 2]
damerau-levenstein matrix  2
damerau-levenstein recurent  2
```

Рис. 9: Результат нахождения редакторского расстояния при добавлении символов в первую строку

```
first string to compare: aaaa
second string to compare: aa
string1: ( aaaa )  4
string2: ( aa )  2

[0, 1, 2, 3, 4]
[1, 0, 1, 2, 3]
[2, 1, 0, 1, 2]
levenstein matrix  2

[0, 1, 2, 3, 4]
[1, 0, 1, 2, 3]
[2, 1, 0, 1, 2]
damerau-levenstein matrix  2
damerau-levenstein recurent  2
```

Рис. 10: Результат нахождения редакторского расстояния при удалении символов из первой строки

```
first string to compare: aaaaa
second string to compare: bbbbb
string1: ( aaaaa )  5
string2: ( bbbbb )  5

[0, 1, 2, 3, 4, 5]
[1, 1, 2, 3, 4, 5]
[2, 2, 2, 3, 4, 5]
[3, 3, 3, 3, 4, 5]
[4, 4, 4, 4, 4, 5]
[5, 5, 5, 5, 5, 5]
levenstein matrix  5

[0, 1, 2, 3, 4, 5]
[1, 1, 2, 3, 4, 5]
[2, 2, 2, 3, 4, 5]
[3, 3, 3, 3, 4, 5]
[4, 4, 4, 4, 4, 5]
[5, 5, 5, 5, 5, 5]
damerau-levenstein matrix  5
damerau-levenstein recurent  5
```

Рис. 11: Результат нахождения редакторского расстояния при замене символов первой строки

```
first string to compare: stroka
second string to compare: stroka
string1: ( stroka )  6
string2: ( stroka )  6

[0, 1, 2, 3, 4, 5, 6]
[1, 0, 1, 2, 3, 4, 5]
[2, 1, 0, 1, 2, 3, 4]
[3, 2, 1, 0, 1, 2, 3]
[4, 3, 2, 1, 0, 1, 2]
[5, 4, 3, 2, 1, 0, 1]
[6, 5, 4, 3, 2, 1, 0]
levenstein matrix  0

[0, 1, 2, 3, 4, 5, 6]
[1, 0, 1, 2, 3, 4, 5]
[2, 1, 0, 1, 2, 3, 4]
[3, 2, 1, 0, 1, 2, 3]
[4, 3, 2, 1, 0, 1, 2]
[5, 4, 3, 2, 1, 0, 1]
[6, 5, 4, 3, 2, 1, 0]
damerau-levenstein matrix  0
damerau-levenstein recurent  0
```

Рис. 12: Результат нахождения редакторского расстояния при совпадении символов в первой строке

```
first string to compare: forcefully
second string to compare: radically
string1: ( forcefully )  10
string2: ( radically )  9

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1, 1, 2, 2, 3, 4, 5, 6, 7, 8, 9]
[2, 2, 2, 3, 3, 4, 5, 6, 7, 8, 9]
[3, 3, 3, 3, 4, 4, 5, 6, 7, 8, 9]
[4, 4, 4, 4, 4, 5, 5, 6, 7, 8, 9]
[5, 5, 5, 5, 4, 5, 6, 6, 7, 8, 9]
[6, 6, 6, 6, 5, 5, 6, 7, 7, 8, 9]
[7, 7, 7, 7, 6, 6, 6, 7, 7, 7, 8]
[8, 8, 8, 8, 7, 7, 7, 7, 7, 7, 8]
[9, 9, 9, 9, 8, 8, 8, 8, 8, 8, 7]
levenstein matrix  7

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1, 1, 2, 2, 3, 4, 5, 6, 7, 8, 9]
[2, 2, 2, 3, 3, 4, 5, 6, 7, 8, 9]
[3, 3, 3, 3, 4, 4, 5, 6, 7, 8, 9]
[4, 4, 4, 4, 4, 5, 5, 6, 7, 8, 9]
[5, 5, 5, 5, 4, 5, 6, 6, 7, 8, 9]
[6, 6, 6, 6, 5, 5, 6, 7, 7, 8, 9]
[7, 7, 7, 7, 6, 6, 6, 7, 7, 7, 8]
[8, 8, 8, 8, 7, 7, 7, 7, 7, 7, 8]
[9, 9, 9, 9, 8, 8, 8, 8, 8, 8, 7]
damerau-levenstein matrix  7
damerau-levenstein recurent  7
```

Рис. 13: Результат нахождения редакторского расстояния

```
first string to compare: forest
second string to compare: frots
string1: ( forest )  6
string2: ( frots )  5

[0, 1, 2, 3, 4, 5, 6]
[1, 0, 1, 2, 3, 4, 5]
[2, 1, 1, 1, 2, 3, 4]
[3, 2, 1, 2, 2, 3, 4]
[4, 3, 2, 2, 3, 3, 3]
[5, 4, 3, 3, 3, 3, 4]
levenstein matrix  4

[0, 1, 2, 3, 4, 5, 6]
[1, 0, 1, 2, 3, 4, 5]
[2, 1, 1, 1, 2, 3, 4]
[3, 2, 1, 1, 2, 3, 4]
[4, 3, 2, 2, 2, 3, 3]
[5, 4, 3, 3, 3, 2, 3]
damerau-levenstein matrix  3
damerau-levenstein recurent  3
```

Рис. 14: Сравнение нахождения редакторского расстояния Левенштейна и Дамерау-Левенштейна

4.2. Сравнительный анализ на материале экспериментальных данных

В ходе эксперимента были полученны следующие данные:

```
first string to compare: zaqwsx
second string to compare: zaq
string1: ( zaqwsx )  6
string2: ( zaq )  3

[0, 1, 2, 3, 4, 5, 6]
[1, 0, 1, 2, 3, 4, 5]
[2, 1, 0, 1, 2, 3, 4]
[3, 2, 1, 0, 1, 2, 3]
levenstein matrix  3

[0, 1, 2, 3, 4, 5, 6]
[1, 0, 1, 2, 3, 4, 5]
[2, 1, 0, 1, 2, 3, 4]
[3, 2, 1, 0, 1, 2, 3]
damerau-levenstein matrix  3
damerau-levenstein recurent  3

Memory recurse: 19346 + (memory in stack for 1 call)*565
Memory not recurse: 423 + (memory in stack for 1 call)*1
```

Рис. 15: Использование памяти разными реализациями алгоритма. Пример 1

```
first string to compare: zzzzzzzzzzzzz
second string to compare: xx
string1: ( zzzzzzzzzzzzz )  13
string2: ( xx )  2

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
[1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
[2, 2, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
levenstein matrix  13

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
[1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
[2, 2, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
damerau-levenstein matrix  13
damerau-levenstein recurent  13

Memory recurse: 19682 + (memory in stack for 1 call)*547
Memory not recurse: 597 + (memory in stack for 1 call)*1
```

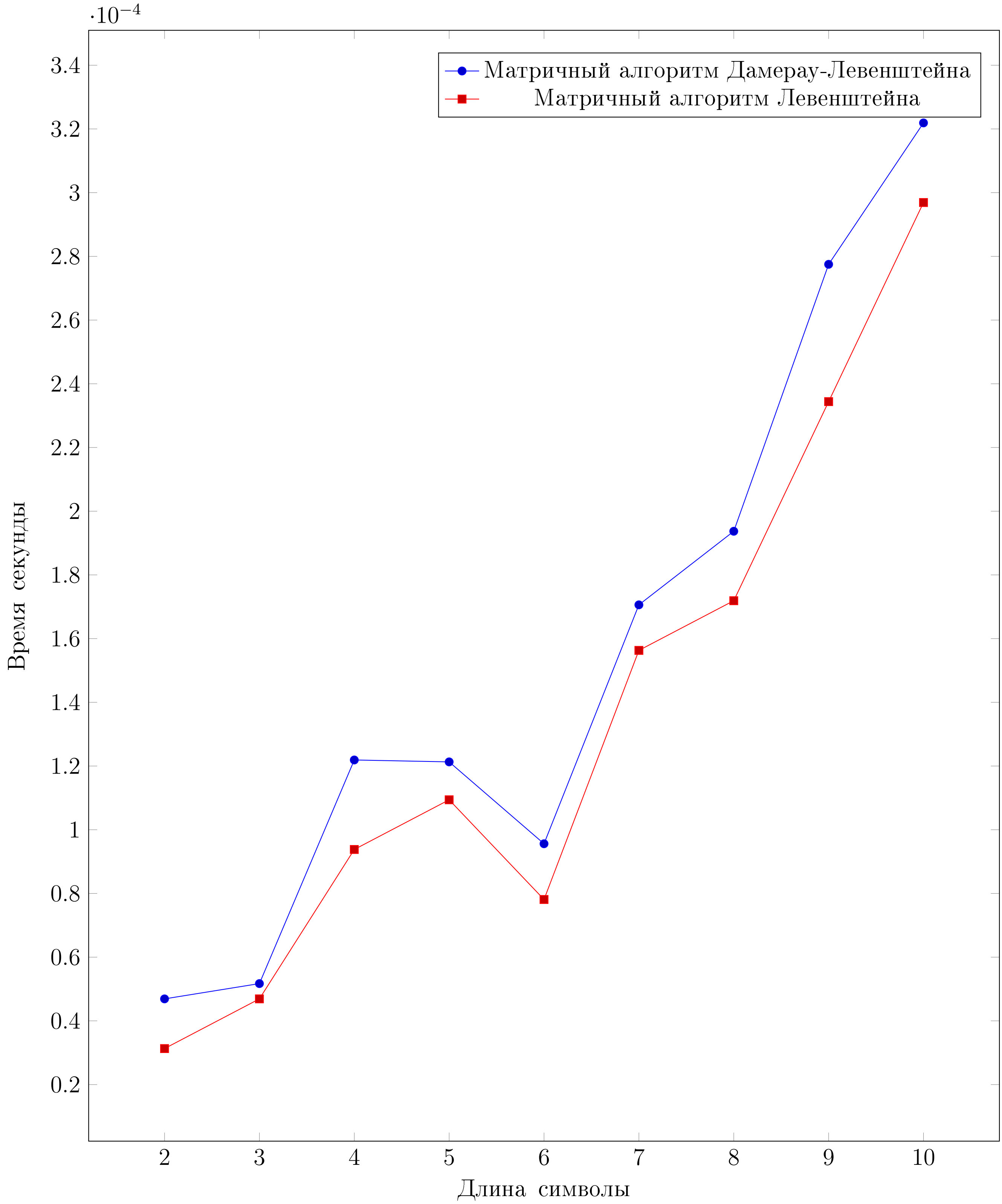
Рис. 16: Использование памяти разными реализациями алгоритма. Пример 2

```
first string to compare: zzzzzzzzzzzzz
second string to compare: xxx
string1: ( zzzzzzzzzzzzz )  13
string2: ( xxx )  3

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
[1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
[2, 2, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
[3, 3, 3, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
levenstein matrix  13

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
[1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
[2, 2, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
[3, 3, 3, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
damerau-levenstein matrix  13
damerau-levenstein recurent  13

Memory recurse: 175110 + (memory in stack for 1 call)*4954
Memory not recurse: 766 + (memory in stack for 1 call)*1
```

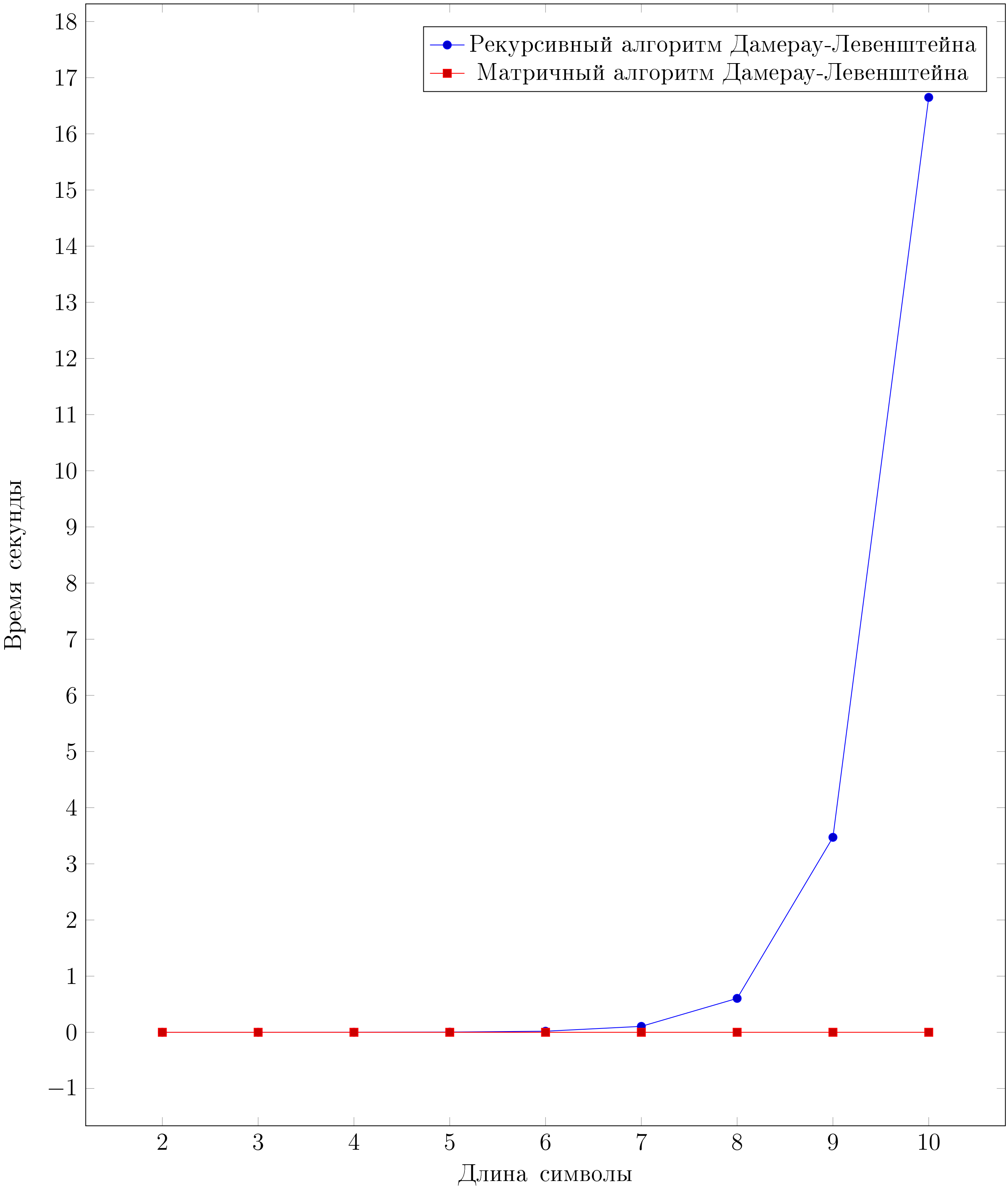
Рис. 17: Использование памяти разными реализациями алгоритма. Пример 3

```
first string to compare: zzzzzzzzzzzzz
second string to compare: xxxx
string1: ( zzzzzzzzzzzzz )  13
string2: ( xxxx )  4

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
[1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
[2, 2, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
[3, 3, 3, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
[4, 4, 4, 4, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
levenstein matrix  13

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
[1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
[2, 2, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
[3, 3, 3, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
[4, 4, 4, 4, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
damerau-levenstein matrix  13
damerau-levenstein recurent  13

Memory recurse: 1183052 + (memory in stack for 1 call)*33853
Memory not recurse: 935 + (memory in stack for 1 call)*1
```

Рис. 18: Использование памяти разными реализациями алгоритма. Пример 4



$\cdot 10^{-4}$
Матричный алгоритм Дамерау-Левенштейна
Матричный алгоритм Левенштейна
Время секунды
Длина символы
3.4
3.2
3
2.8
2.6
2.4
2.2
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
2
3
4
5
6
7
8
9
10



4.3. Вывод из экспериментов

Рекурсивный алгоритм проще для реализации, но на длинных входных строках (в рамках лабораторной для этого алгоритма строки уже длиной 9 являлись длительно обрабатываемыми) оказывается неэффективным как по памяти, так и времени выполнения. Матричная реализация данного алгоритма в аналогичных условиях потребляет в разы меньше памяти и работает быстрее.

Заключение

В результате выполнения данной работы рассмотренны и изучены понятия расстояния Левенштейна и расстояния Дамерау-Левенштейна. Реализован матричный вариант алгоритма нахождения расстояния Левенштейна. Реализовано два варианта алгоритма нахождения расстояния Дамерау-Левенштейна (в рекурсивной и матричной формах). Сравнены их временные характеристики как следствие проведённых экспериментов. Были сделаны выводы об эффективности по времени рекурсивного и нерекурсивного вариантов алгоритмов.