МГТУ им. Баумана

Лабораторная работа №1

По курсу: "Анализ алгоритмов"

Расстояние Левенштейна и Дамерау-Левенштейна

Студент: Аникин И. А., ИУ7-51Б

Преподаватель: Волкова Л.Л.

Москва, 2020

Содержание

Введение

Строковый тип данных - неотъемлемая составляющая программирования. Большое количество информации удобнее всего представлять именно в текстовом виде.
Числа имеют определенный порядок, а значит их можно сравнивать и мы легко можем определить является ли число меньше или больше заданного. Но для строк нет единого соглашения о том, как измерять разность строк. Есть лексикографический порядок, но он учитывает только буквенный порядок.
Решение данной проблемы придумал советский математик Левенштейн. Он изобрел собственную метрику, которую впоследствии назвали расстоянием Левенштейна. Она определяется как минимальное количество односимвольных операций вставки, замены и удаления, которое переводит одну последовательность символов в другую и показывает насколько эти две последовательности символов отличаются друг от друга. Вводится понятие редакторской операции. Редакторская операция - это операция, которая может быть применена к любому символу строки. Расстояние Левенштейна учитывает 4 различных действия: вставка символа (insert), удаление символа (delete), замена символа (replace) и совпадение символа (match). Обычно они обозначаются первыми буквами английских названий. У каждой операции есть число, которое называется штрафом. Для вставки, замены и удаления они равны 1, для совпадения штраф равен 0. В общем случае, можно назначить каждой операции разный штраф в зависимости от задачи. Расстояние - это сумма штрафов всех операций.
Расстояние Дамерау-Левенштейна является развитием идеи расстояния Левенштейна. Для вычисления этого расстояния вводится еще одна редакторская операция - транспозиция (перестановка двух рядом стоящих символов).
Данные метрики очень широко применяются в компьютерной лингвистике и биоинформатике. Например, при вводе с клавиатуры можно исправить опечатки, если заменять набранный текст максимально похожим словом из словаря. Еще одним применением является сравнение генов, хромосом и белков. Части цепочек молекул кодируются буквами и таким образом можно сравнить насколько отличаются две цепочки молекул.

1. Аналитическая часть

Рассмотрим цели и задачи данной лабораторной работы.
Цель лабораторной работы - разработать и провести сравнительный анализ алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.
Для достижения данной цели можно выделить следующие задачи:

- разработать алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна;
- реализовать разработанные алгоритмы;
- провести тестирование разработанных методов по методу черного ящика;
- провести замеры времени работы реализации алгоритма на наборе данных;
- провести теоретическую оценку максимально потребляемого объема памяти;
- провести сравнительный анализ временной и емкостной эффективности реализаций алгоритма.

Дадим определение расстоянию Левенштейна.
Расстояние Левенштейна - это число, которое показывает какое количество редакторских операций (вставки, удаления и замены символов) надо совершить, что перевести одну последовательность символов в другую. Вычислить это расстояние можно следующим образом.
Пусть S_1 и S_2 - две строки с длинами N и M, $d(S_1, S_2)$ - расстояние Левенштейна.
$d(S_1, S_2) = D(N, M)$, где

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

где $m(a, b)$ равна нулю, если $a = b$ и равна единице иначе. В данном рекуррентном соотношении шаг по индексу i символизирует удаление из первой строки, по j - вставку в первую строку, а шаг по обоим индексам символизирует замену символа при несовпадении и отсутствие изменений иначе.
Расстояние Дамерлау-Левенштейна добавляет еще одну редакторскую операцию: транспозицию. Штраф данной операции также равен 1. Расстояние

Дамерлау-Левенштейна можно вычислять с помощью следующего рекуррентного соотношения:
Пусть S_1 и S_2 - две строки с длинами N и M, $d(S_1, S_2)$ - расстояние Дамерау-Левенштейна.
$d(S_1, S_2) = D(N, M)$, где

$$D(N,M) = \begin{cases} 0, & i = 0, j = 0, \\ i, & i > 0, j = 0, \\ j, & i = 0, j > 0, \\ min(D(i,j-1)+1, & \\ \quad D(i-1,j)+1, & i > 1, j > 1, \\ \quad D(i-1,j-1)+m(S_1[i],S_2[j]), & S_{1i} = S_{2j-1}, S_{2j} = S_{1i-1}. \\ \quad D(i-2,j-2)+1), & \\ min(D(i,j-1)+1, & \\ \quad D(i-1,j)+1, & i > 0, j > 0, \\ \quad D(i-1,j-1)+m(S_1[i],S_2[j]), & \end{cases}$$

Здесь шаги соответствует тем же операциям, что и в расстоянии Левенштейна, а дополнительный шаг $(D(i-2, j-2)+1)$ представляет собой транспозицию.

2. Конструкторская часть

2.1. Требования к ПО

Разработанное ПО должно обладать следующей функциональностью:

- считывать 2 строки,
- выводить расстояние и (кроме рекурсивного алгоритма) матрицу,
- реализовать поиск расстояния Левенштейна и Дамерау-Левенштейна,
- осуществлять замеры процессорного времени работы реализаций алгоритмов.

В ходе данной лабораторной работы будет реализовано 3 различных алгоритма для поиска расстояния Левенштейна и 1 алгоритм поиска расстояния Дамерау-Левенштейна.
Для поиска расстояния Левенштейна будут реализованы следующие алгоритмы:

- нерекурсивный матричный алгоритм;
- рекурсивный алгоритм;
- рекурсивный матричный алгоритм.

Рекурсивный матричный алгоритм является улучшением рекурсивного алгоритма и использует матрицу для сохранения ранее вычисленных значений, чтобы не осуществлять повторных вычислений.
Для поиска расстояния Дамерау-Левенштейна будет реализован нерекурсивный матричный алгоритм.
На схемах алгоритмов используются следующие обозначения: s1, s2 - строки символов; i, j - текущие длины рассматриваемых строк; M - матрица с расстояниями. На рисунке 2 представлена схема нерекурсивного матричного алгоритма.
На рисунке 3 представлена схема рекурсивного алгоритма.
На рисунке 4 представлена схема рекурсивного матричного алгоритма.
На рисунке 5 представлена схема матричного алгоритма поиска расстояния Дамерау-Левенштейна.

3. Технологическая часть

В ходе выполнения лабораторной работы были реализованы следующие модули:

- модуль levenshtein.py, в котором реализовано 3 алгоритма поиска расстояний Левенштейна и 1 алгоритм поиска расстояния Дамерау-Левенштейна;
- модуль test.py, предназначенный для тестирования реализованных алгоритмов;
- модуль analysis.py, предназначенный для замеров времени выполнения алгоритмов на входных данных различной длины и построения графиков;
- модуль main.py, предоставляющий пользователю возможности работы с функциями поиска расстояний.

В качестве языка разработки был выбран язык python, т. к. он имеет ряд преимуществ для достижения цели лабораторной работы:

- высокая скорость разработки программ;
- простой синтаксис языка;
- наличие большого количества библиотек, что позволяет легко анализировать полученные результаты и проводить тестирование.

3.1. Тестирование

Тестирование разработанных алгоритмов проводилось по методу черного ящика на наборе тестов, который представлен в таблице 1.

Таблица 1: Набор тестов

Входные данные		Ожидаемые выходные данные	
s1	s2	Расстояние Л.	Расстояние Д.-Л.
""	"teststring"	10	10
"notteststring"	""	13	13
"IAmEqualTo"	"IAmEqualTo"	0	0
"telo"	"stolb"	3	3
"polynomial"	"exponential"	6	6
"abba"	"abab"	2	1

Все тесты пройдены успешно.

3.2. Листинги алгоритмов

На листинге 1 приведен алгоритм поиска расстояния Левенштейна матричным методом.

Листинг 1: Нерекурсивный матричный алгоритм

```
def levenshtein_matrix(s1, s2):
    n1 = len(s1)
    n2 = len(s2)
    matrix = [[0 for j in range(n2+1)] for i in range(n1+1)]

    for i in range(n1+1):     #i = 0, j > 0
        matrix[i][0] = i

    for j in range(n2+1):     #j > 0, i = 0
        matrix[0][j] = j

    for i in range(1, n1+1):        #i > 0, j > 0
        for j in range(1, n2+1):
            m = 0 if s1[i-1] == s2[j-1] else 1
            matrix[i][j] = min(matrix[i-1][j] + 1,
                               matrix[i][j-1] + 1,
                               matrix[i-1][j-1] + m)

    print_matrix(s1, s2, matrix)

    return matrix[-1][-1]
```

На листинге 2 приведен алгоритм поиска расстояния Левенштейна рекурсивным методом.

Листинг 2: Рекурсивный алгоритм

```
def levenshtein_recursive(s1, s2, i, j):
    if min(i, j) == 0:
        return max(i, j)
    else:
        m = 0 if s1[i-1] == s2[j-1] else 1
        return min(levenshtein_recursive(s1, s2, i-1, j) + 1,
                   levenshtein_recursive(s1, s2, i, j-1) + 1,
                   levenshtein_recursive(s1, s2, i-1, j-1) + m)
```

На листинге 3 приведен алгоритм поиска расстояния Левенштейна рекурсивным матричным методом.

Листинг 3: Рекурсивный матричный алгоритм

```
    if min(i, j) == 0:
```

```
        matrix[i][j] = max(i, j)
    else:
        if matrix[i][j] == float("inf"):
            m = 0 if s1[i-1] == s2[j-1] else 1
            matrix[i][j] = min(levenshtein_recursive_matrix(s1, s2, i
                -1, j, matrix) + 1,
                               levenshtein_recursive_matrix(s1, s2, i
                                   , j-1, matrix) + 1,
                               levenshtein_recursive_matrix(s1, s2, i
                                   -1, j-1, matrix) + m)
    return matrix[i][j]
```

На листинге 4 приведен алгоритм поиска расстояния Дамерау-Левенштейна матричным методом.

Листинг 4: Матричный алгоритм поиска расстояния Дамерау-Левенштейна

```
def dameray_levenshtein(s1, s2):
    n1 = len(s1)
    n2 = len(s2)
    matrix = [[0 for j in range(n2+1)] for i in range(n1+1)]

    for i in range(n1+1):     #i = 0, j > 0
        matrix[i][0] = i

    for j in range(n2+1):     #j > 0, i = 0
        matrix[0][j] = j

    for i in range(1, n1+1):        #i > 0, j > 0
        for j in range(1, n2+1):
            m = 0 if s1[i-1] == s2[j-1] else 1
            if i > 1 and j > 1 and s1[i-2] == s2[j-1] and s1[i-1] ==
                s2[j-2]:
                matrix[i][j] = min(matrix[i-1][j] + 1,
                                   matrix[i][j-1] + 1,
                                   matrix[i-1][j-1] + m,
                                   matrix[i-2][j-2] + 1)
            else:
                matrix[i][j] = min(matrix[i-1][j] + 1,
                                   matrix[i][j-1] + 1,
                                   matrix[i-1][j-1] + m)
    print_matrix(s1, s2, matrix)

    return matrix[-1][-1]
```

4. Исследовательская часть

Проведем исследование временных и емкостных характеристик реализованных алгоритмов.
Замеры времени проводились с использованием следующего аппаратного обеспечения: i5-8265U, 6 ГБ ОЗУ.
На рисунке 1 представлены графики зависимости времени работы алгоритмов от длины входных данных. На вход подаются две строки одинаковой длины.

Рис. 1: Сравнение временных характеристик алгоритмов

Исходя из данных графиков видно, что самым быстрым является нерекурсивный матричный алгоритм поиска расстояния Левенштейна. Чуть медленнее работает нерекурсивный матричный алгоритм поиска расстояния Дамерау-Левенштейна из-за дополнительной редакторской операции. Далее идет рекурсивный матричный алгоритм, который является более медленным из-за использования рекурсивных вызовов. Самым медленным является рекурсивный алгоритм из-за большого количества повторяющихся рекурсивных вызовов.
Проведем теоретическую оценку реализованных алгоритмов по затрачиваемой

памяти.
В нерекурсивном матричном алгоритме используется целочисленная матрица, состоящая из $n+1$ строк и $m+1$ столбца, где n - длина строки s_1, m - длина строки s_2.
Размер целочисленной переменной равен 4 байта. Следовательно, матрица занимает $4(n+1)(m+1)$ байт.
Помимо матрицы используются две целочисленные переменные, хранящие размеры строк, две целочисленные переменные-счетчики в циклах и одна переменная, использующаяся для сравнения символов. Итого, нерекурсивный матричный алгоритм требует $4(n+1)(m+1)+20$ байт памяти.
В рекурсивном алгоритме каждый рекурсивный вызов занимает 8 байт на адрес возврата (при использовании 64-разрядной машины), 4 байта на переменную для результата, две ссылки на строки по 8 байт, две переменные, содержащие длину строк по 4 байта и одна локальная переменная, использующаяся для сравнения символов. Итого, на один рекурсивный вызов необходимо 40 байт. Максимальное количество вызовов равно $n+m+1$. Итого, алгоритм требует $40(n+m+1)$ байт памяти.
В рекурсивном матричном алгоритме используется матрица, аналогичная нерекурсивному матричному методу. Она занимает $4(n+1)(m+1)$ байт. Один рекурсивный вызов также занимает 40 байт. Максимально происходит n рекурсивных вызовов. Итого алгоритм требует байт $4(n+1)(m+1)+40n$ памяти.
Нерекурсивный матричный алгоритм поиска расстояния Дамерау-Левенштейна занимает также, как и первый алгоритм $4(n+1)(m+1)+20$ байт памяти.

В результате анализа, можно сделать вывод о том, что наименьшее количество памяти потребляет рекурсивный алгоритм. Матричный алгоритм занимает больше памяти, чем рекурсивный, но меньше, чем матричный рекурсивный.

Заключение

В ходе выполнения лабораторной работы были выполнены следующие задачи:

- разработаны алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна;
- реализованы разработанные алгоритмы;
- проведено тестирование разработанных методов по методу черного ящика;
- проведены замеры времени работы реализации алгоритма на наборе данных;
- проведена теоретическая оценка максимально потребляемого объема памяти;
- проведен сравнительный анализ временной и емкостной эффективности реализаций алгоритма.

Список литературы

1) 1. Гасфилд, ?. Строки, деревья и последовательности в алгоритмах. Информатика и вычислительная биология. — СПб.: Невский Диалект БВХ-Петербург, 2003. — 0 с.

2. В. И. Левенштейн. Двоичные коды с исправлением выпадений, вставок и замещений символов. Доклады Академий Наук СССР, 1965.

Приложения

Приложение 1

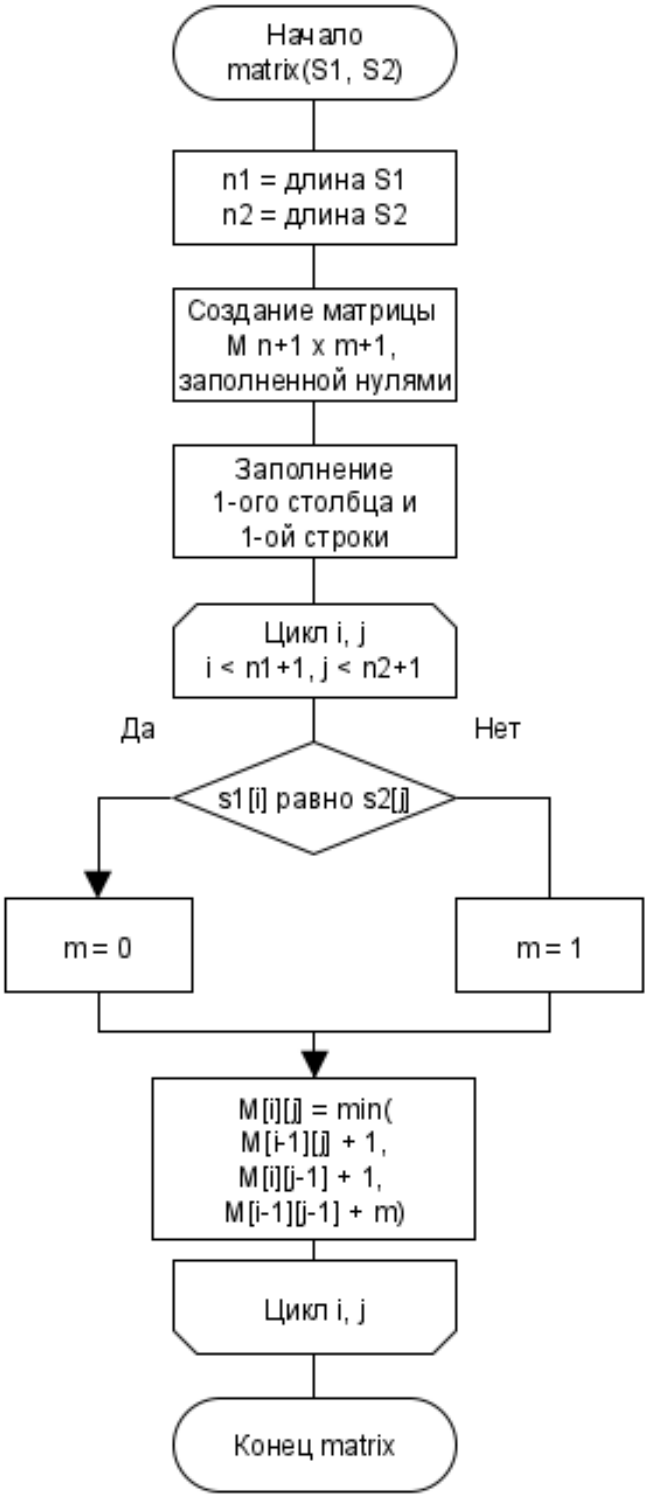

Рис. 2: Схема нерекурсивного матричного алгоритма

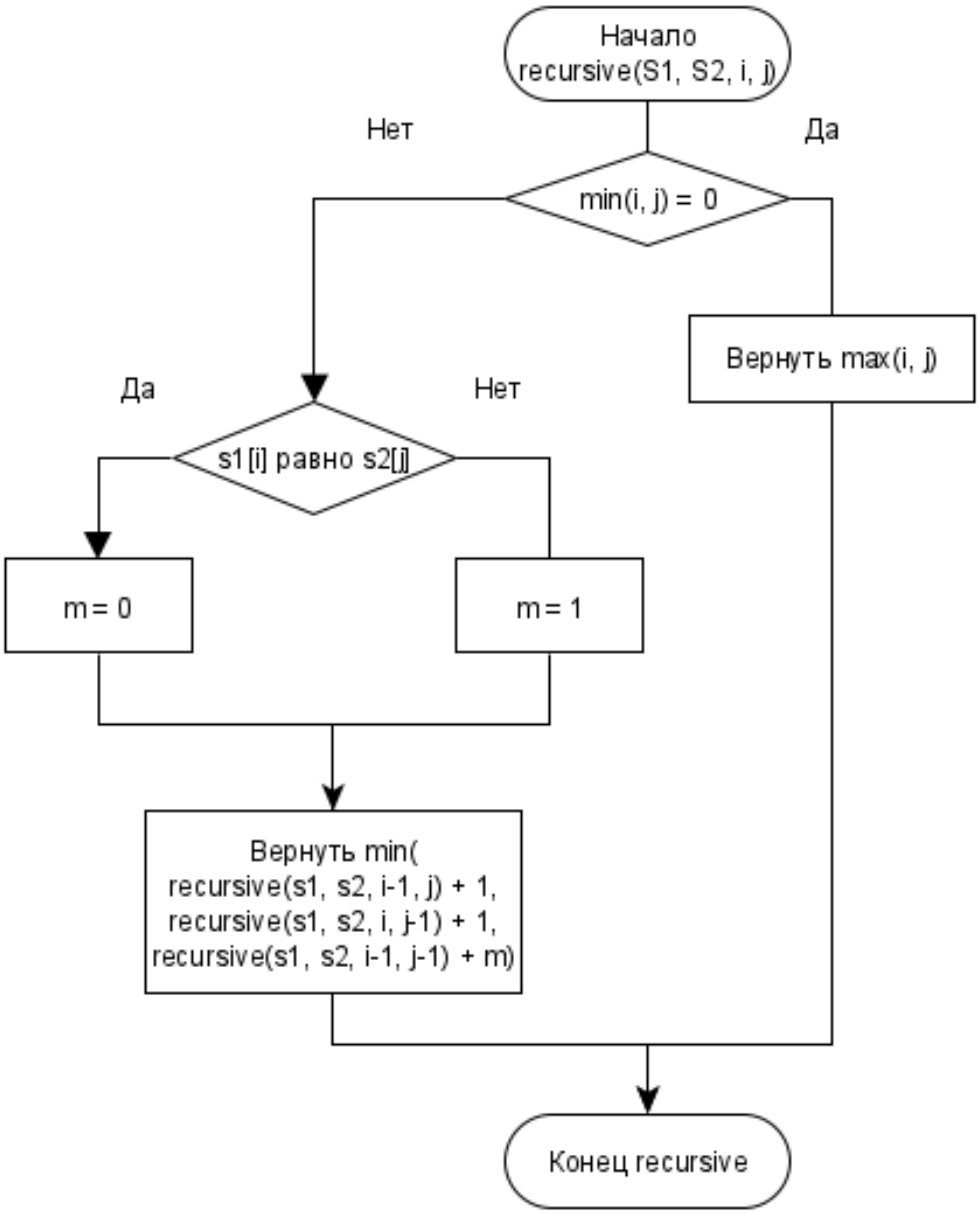

Рис. 3: Схема рекурсивного алгоритма

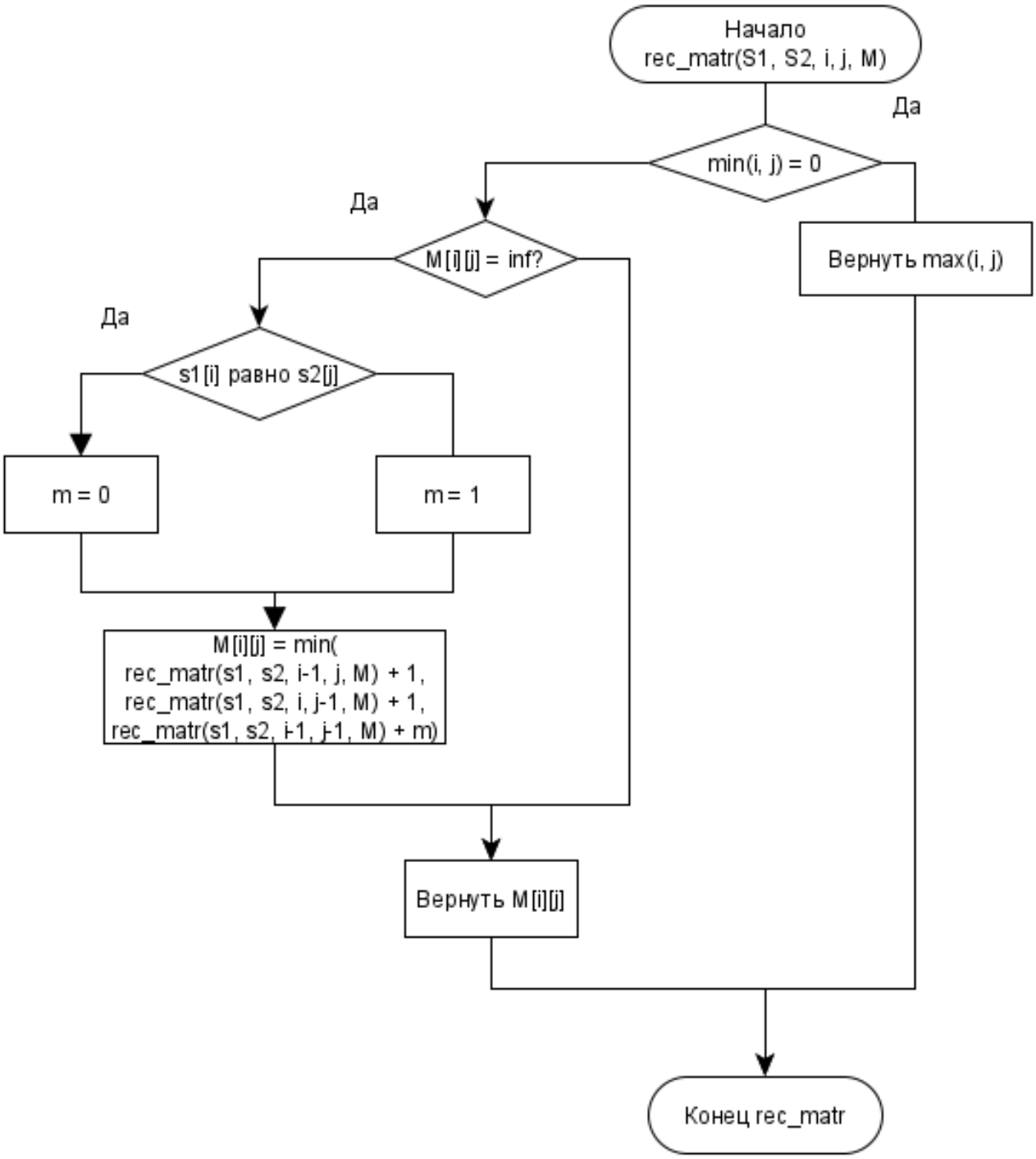

Рис. 4: Схема рекурсивного матричного алгоритма

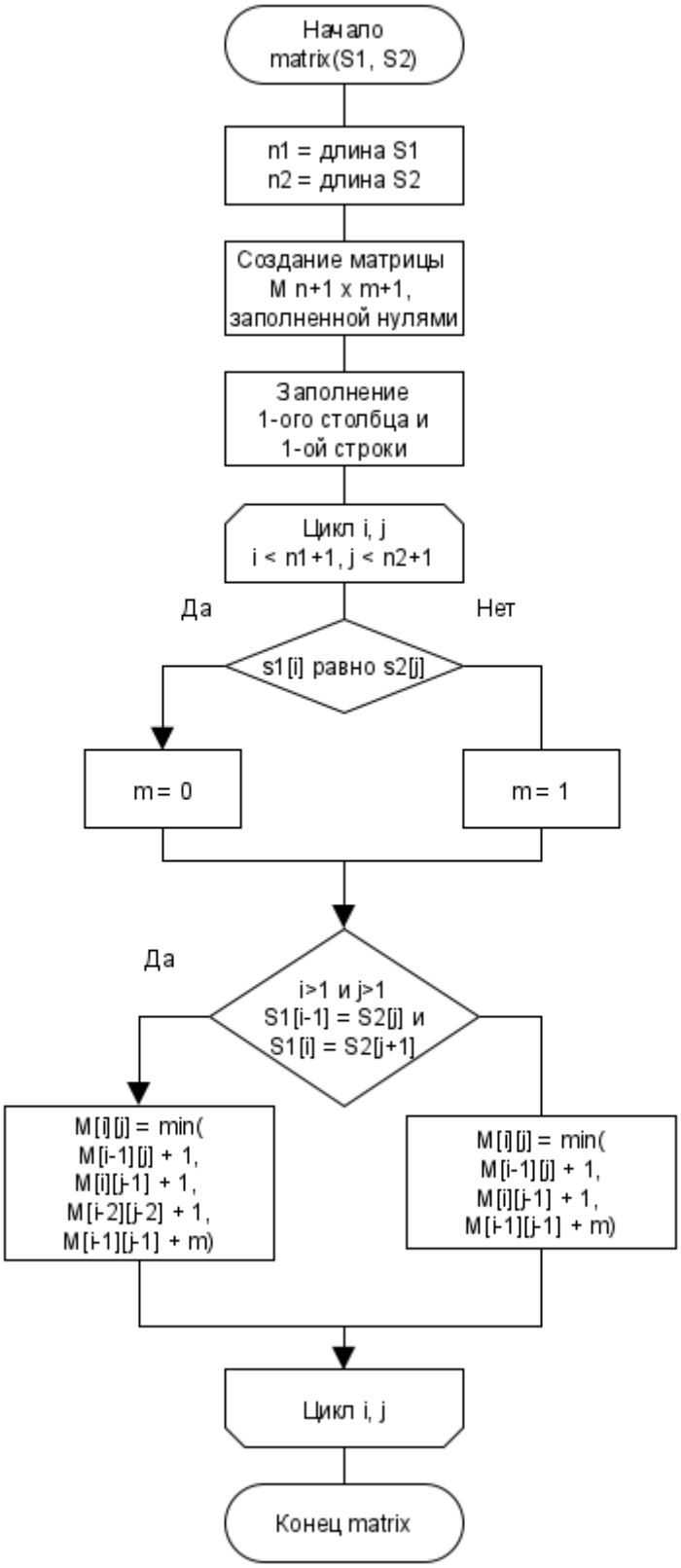

Рис. 5: Схема нерекурсивного матричного алгоритма поиска расстояния Дамерау-Левенштейна