МГТУ им. Баумана

Лабораторная работа №1

По курсу: "Анализ алгоритмов"

Расстояние Левенштейна

Работу выполнил: Лумбунов Дмитрий, ИУ7-54Б

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

Москва, 2019

Оглавление

Введение

Расстояние Левенштейна (также редакционное расстояние или дистанция редактирования) — это минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

Впервые задачу поставил в 1965 году советский математик Владимир Иосифович Левенштейн при изучении последовательностей 0-1. Впоследствии более общую задачу для произвольного алфавита связали с его именем. Большой вклад в изучение вопроса внёс Дэн Гасфилд.

Расстояние Левенштейна и его обобщения активно применяется 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; в биоинформатике для сравнения генов, хромосом и белков.

Расстояние Дамерау — Левенштейна (названо в честь учёных Фредерика Дамерау и Владимира 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. Является модификацией расстояния Левенштейна: к операциям вставки, удаления и замены символов, определённых в расстоянии Левенштейна добавлена операция транспозиции (перестановки) символов. Расстояние Дамерау-Левенштейна, как и метрика Левенштейна, является мерой "схожести"двух строк. Алгоритм его поиска находит применение в реализации нечёткого поиска, а также в биоинформатике (сравнение ДНК), несмотря на то, что изначально алгоритм разрабатывался для сравнения текстов, набранных человеком (Дамерау показал, что 80% человеческих ошибок при наборе текстов составляют перестановки соседних символов, пропуск символа, добавление нового символа, и ошибка в символе. Поэтому метрика Дамерау-Левенштейна часто используется в редакторских программах для проверки правописания).

Задачи работы

Задачами данной лабораторной являются:

1. изучение алгоритмов Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;
2. получение практических навыков реализации указанных алгоритмов: двух алгоритмов в матричной версии и одного из алгоритмов в рекурсивной версии;
3. сравнительный анализ линейной и рекурсивной реализаций выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);
4. 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;
5. описание и обоснование полученных результатов в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

1 Аналитическая часть

В этом разделе содержатся описания алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна и их практическое применение.

1.1 Расстояние Левенштейна

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

Каждая операция имеет свой вес (штраф). Редакционным предписанием называется последовательность действий, необходимых для получения из первой строки вторую, и минимизирующую суммарную цену. Суммарная цена есть искомое расстояние Левенштейна.

Введем следующие обозначения операций:

1. D (delete) — удалить,
2. I (insert) — вставить,
3. R (replace) — заменить,
4. M (match) - совпадение.

Пусть S_1 и S_2 — две строки (длиной M и N соответственно) над некоторым алфавитом, тогда расстояние Левенштейна можно подсчитать по следующей рекуррентной формуле:

$$D(i,j) = \begin{cases} 0, i = 0, j = 0 \\ i, j = 0, i > 0 \\ j, i = 0, j > 0 \\ min \begin{cases} D(s1[1..i], s2[1..j-1]) + 1, \\ D(s1[1..i-1], s2[1..j]) + 1, \\ D(s1[1..i-1], s2[1..j-1]) + (S1[i] == S2[j] ? 0 : 1)) \end{cases} \end{cases}$$

, где 1 - вставка символа, 2 - удаление символа, 3 - замена символа, при этом, если $S1[i-1] = S2[j-1]$, то вес такой операции будет равен 0, иначе 1.

Существует 2 основных способа нахождения расстояния Левенштейна: матричный и рекурсивный.

Для нахождения кратчайшего расстояния матричным способом необходимо вычислить матрицу D, используя вышеприведённую формулу. Матрица размером $(length(S1)+1)$x$((length(S2)+1)$, где $length(S)$ — длина строки S. Значение в ячейке $[i,j]$ равно значению $D(S1[1...i], S2[1...j])$. Первая строка и первый столбец тривиальны. В соответствии с формулой получаем: $A[i][j] = min(A[i-1][j]+1, A[i][j-1]+1, A[i-1][j-1]+m(S1[i], S2[j]))$, где $m(S1[i], S2[j])$ — функция, равная 0 при $S1[i-1] = S2[j-1]$ и 1 в обратном случае. В результате расстоянием Левенштейна будет ячейка матрицы с индексами $i = length(S1)$ и $j = length(S2)$. Для восстановления редакционного предписания требуется вычислить матрицу D, после чего идти из правого нижнего угла в левый верхний, на каждом шаге находя минимальное из трёх значений:

- если минимально ($D(i-1,j)$ + цена удаления символа $S1[i]$), добавляем удаление символа $S1[i]$ и идём в $[i-1,j]$;
- если минимально ($D(i,j-1)$ + цена вставки символа $S2[j]$), добавляем вставку символа $S2[j]$ и идём в $[i,j-1]$;
- если минимально ($D(i-1,j-1)$ + цена замены символа $S1[i]$ на символ $S2[j]$), добавляем замену $S1[i]$ на $S2[j]$ (если они не равны; иначе ничего не добавляем), после чего идём в $[i-1,j-1]$.

Второй способ нахождения расстояния Левенштейна - рекурсивно в соответствии с формулами, обрабатывая подстроки, пока не будет достигнут тривиальный случай. Работать с подстроками затратно по объему памяти, также в рекурсивном алгоритме будут обрабатываться одинаковые случаи несколько раз, что неэффективно по памяти и по времени.

1.2 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. Является модификацией расстояния Левенштейна: к операциям вставки, удаления и замены символов, определённых в расстоянии Левенштейна добавлена операция транспозиции (перестановки) символов.

Расстояние Дамерау-Левенштейна вычисляется по следующей рекуррентной формуле:

$$D(i,j) = \begin{cases} 0, i=0, j=0 \\ i, j=0, i>0 \\ j, i=0, j>0 \\ min \begin{cases} D(s1[1..i], s2[1..j-1]) + 1, \\ D(s1[1..i-1], s2[1..j]) + 1, \\ D(s1[1..i-1], s2[1..j-1]) + (S1[i] == S2[j] ? 0 : 1)), \\ D(s1[1..i-2], s2[1..j-2]) + 1, \\ \qquad \text{if } i,j > 1 \text{ and } s1[i] = s2[j-1], s1[i-1] = s2[j] \end{cases} \end{cases}$$

, где 1 - вставка символа, 2 - удаление символа, 3 - замена символа, при этом, если $S1[i-1] = S2[j-1]$, то вес такой операции будет равен 0, иначе 1,
4 - перестановка символов в случае, если $S1[i-1] = S2[j]$ и $S1[i] = S2[j-1]$.

Решения задачи нахождения расстояния Дамерау-Левенштейна аналогичны решениям задачи нахождения расстояния Левенштейна.

При матричной реализации формула для ячейки [i,j] модифицируется следующим образом: $A[i][j] = min(A[i-1][j]+1, A[i][j-1]+1, A[i-1][j-1]+m(s1[i], s2[j]), A[i-2][j-2]+1$, если $s1[i] = s2[j-1]$ и $s1[i-1] = s2[j]$ и $i > 1$ и $j > 1$. Для восстановления редакционного предписания требуется вычислить матрицу D, после чего идти из правого нижнего угла в левый верхний, на каждом шаге ища минимальное из четырех значений:

- если минимально ($D(i-1, j)$ + цена удаления символа $S1[i]$), добавляем удаление символа $S1[i]$ и идём в $[i-1, j]$;

- если минимально ($D(i, j-1)$ + цена вставки символа $S2[j]$), добавляем вставку символа $S2[j]$ и идём в $[i, j-1]$;

- если минимально ($D(i-1, j-1)$ + цена замены символа $S1[i]$ на символ $S2[j]$), добавляем замену $S1[i]$ на $S2[j]$ (если они не равны; иначе ничего не добавляем), после чего идём в $[i-1, j-1]$

- если транспозиция возможна, то возвращаем ($D(i-2, j-2)+1$

При рекурсивном решении добавляется еще одна ветвь в дерево рекурсий,когда проверяется условие того, что два заключительных символа очередной подстроки равны двум другим из второй подстроки с учетом транспозиции.

1.3 Практическое применение

Данные алгоритмы применяются при поиске информации по запросу, с помощью них можно найти наиболее подходящее(имеющее наименьшее расстояние) к нему слово и заменить его в поисковой строке. Метод оценки редакционного расстояния активно применяется в биоинформатике для сравнения генов, хромосом и белков.

2 Конструкторская часть

В этом разделе содержатся схемы алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна и сравнительный анализ матричной и рекурсивной реализаций.

2.1 Схемы алгоритмов

На Рис.2.1 и 2.2 представлена схема матричного алгоритма нахождения расстояния Левенштейна.

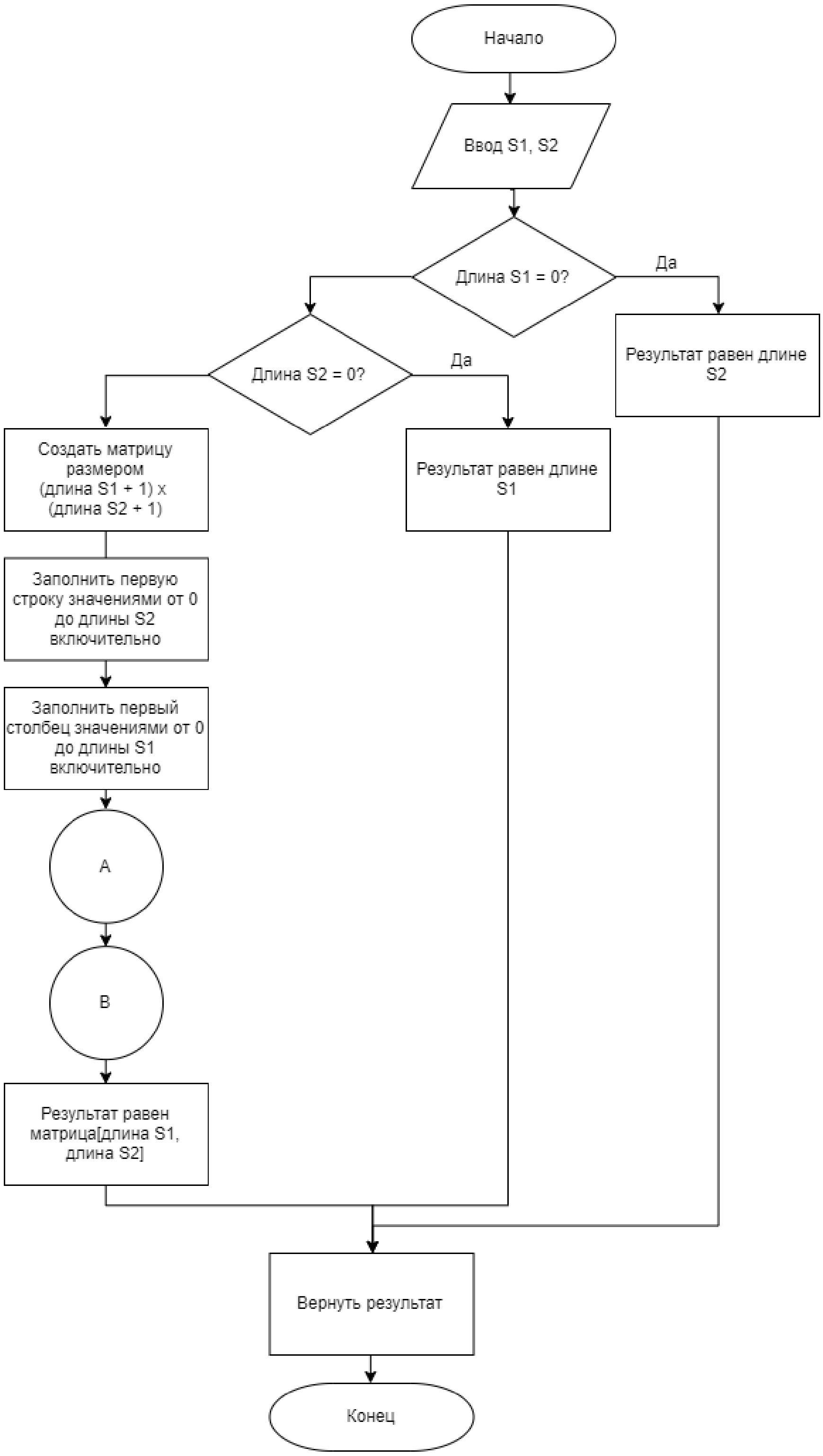

Рис. 2.1: Матричный алгоритм нахождения расстояния Левенштейна

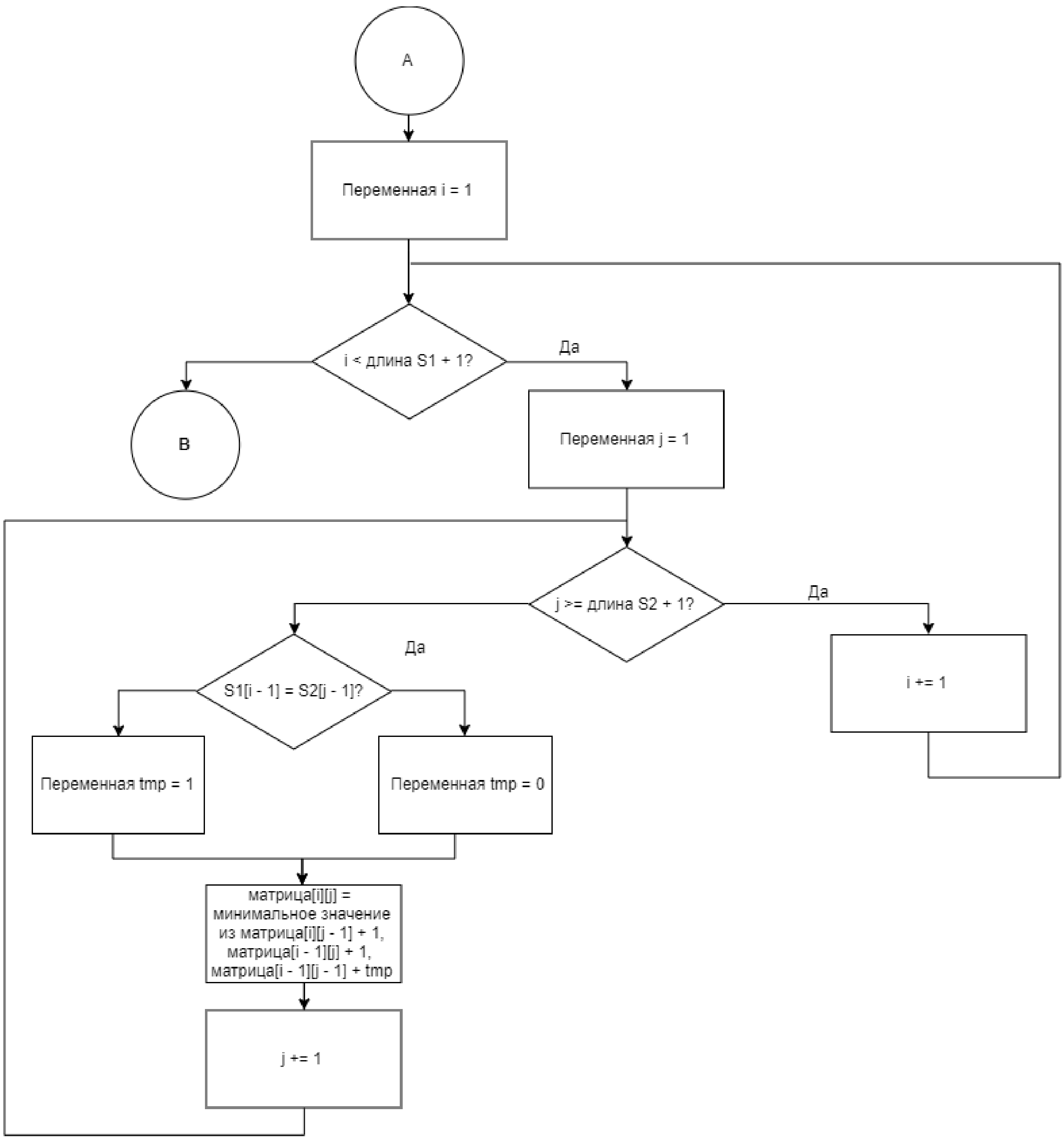

Рис. 2.2: Матричный алгоритм нахождения расстояния Левенштейна (продолжение)

На Рис.2.3 и 2.4 представлена схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна.

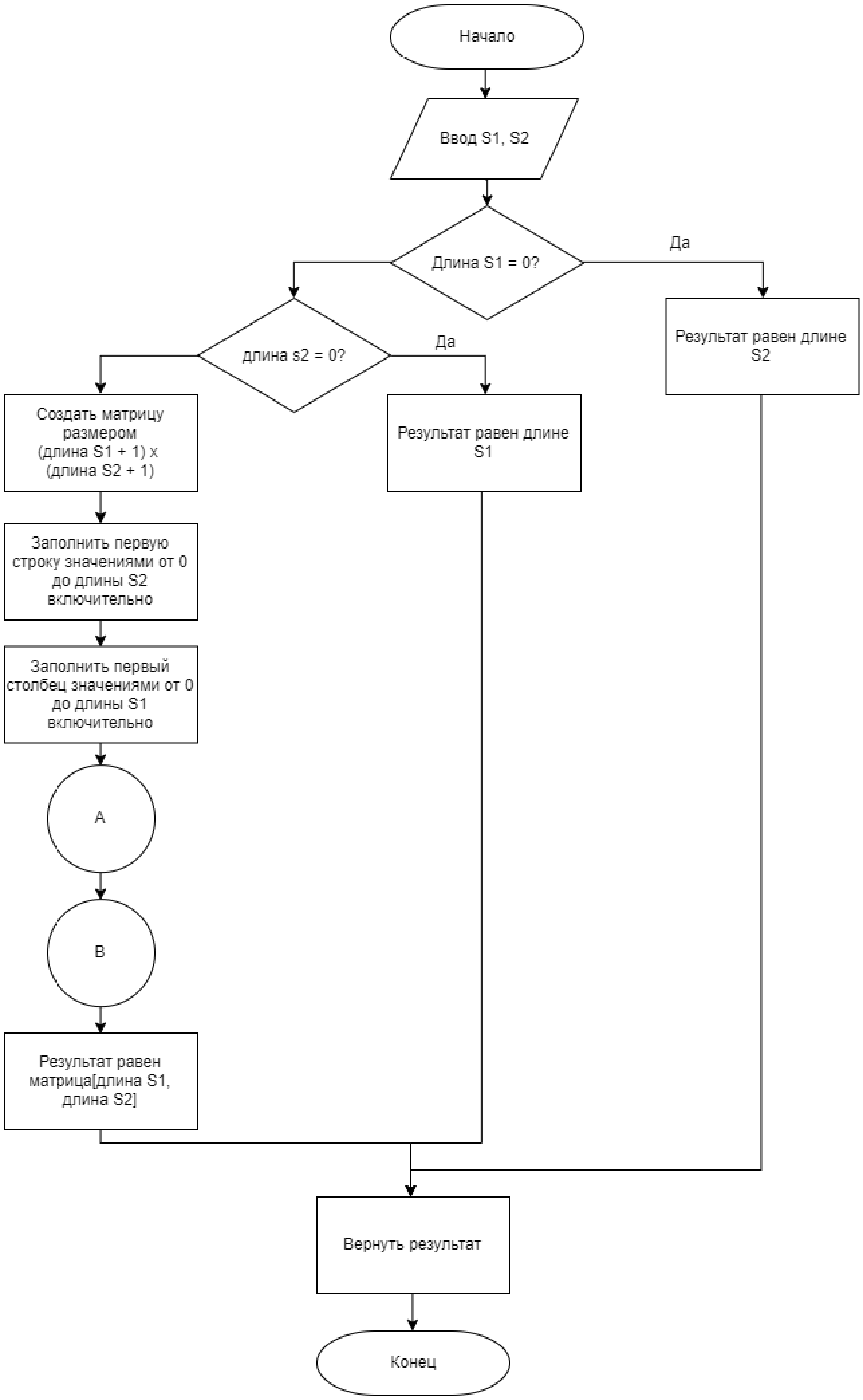

Рис. 2.3: Матричный алгоритм нахождения расстояния Дамерау-Левенштейна

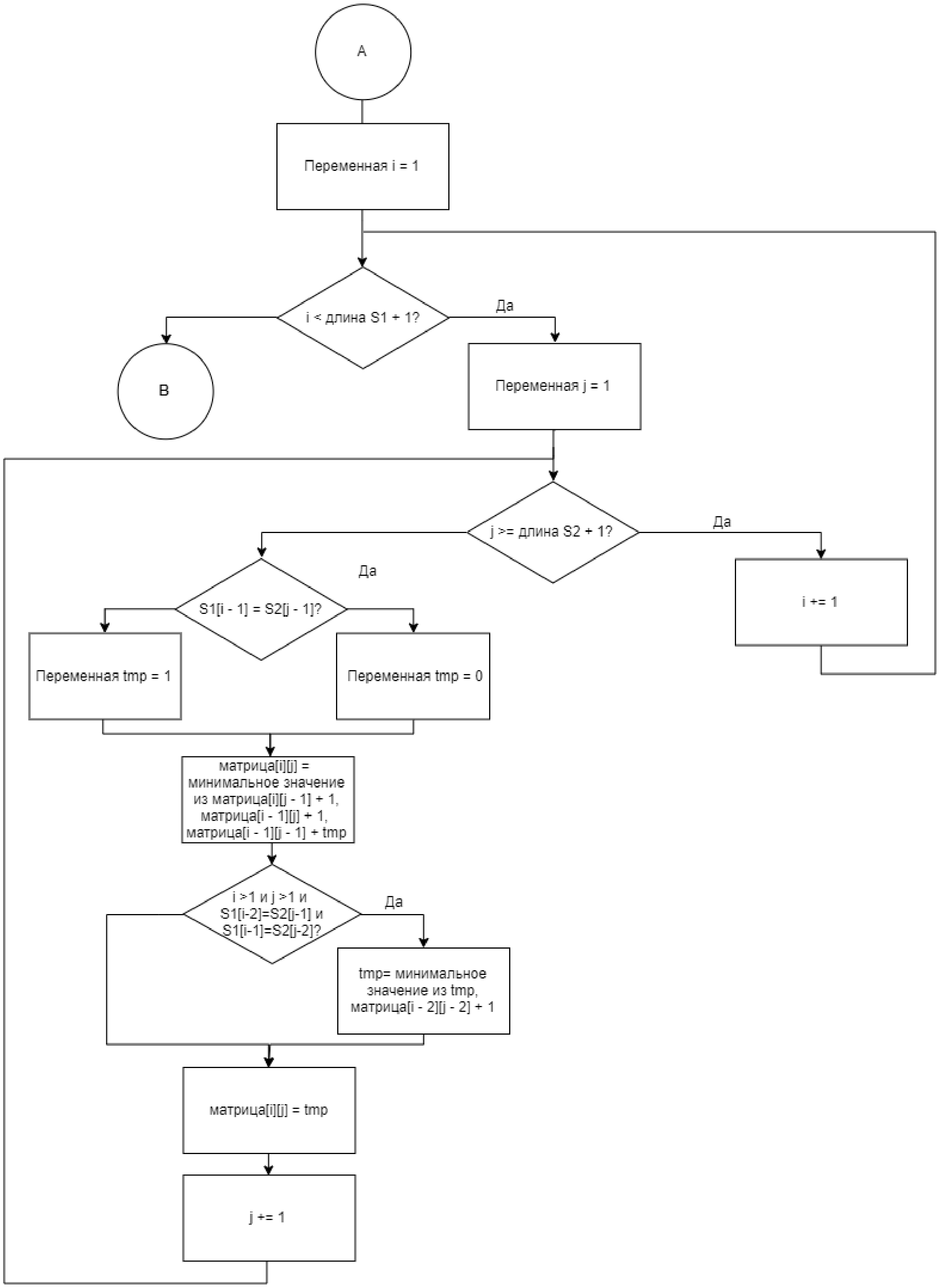

Рис. 2.4: Матричный алгоритм нахождения расстояния Дамерау-Левенштейна (продолжение)

На Рис.2.5 представлена схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна.

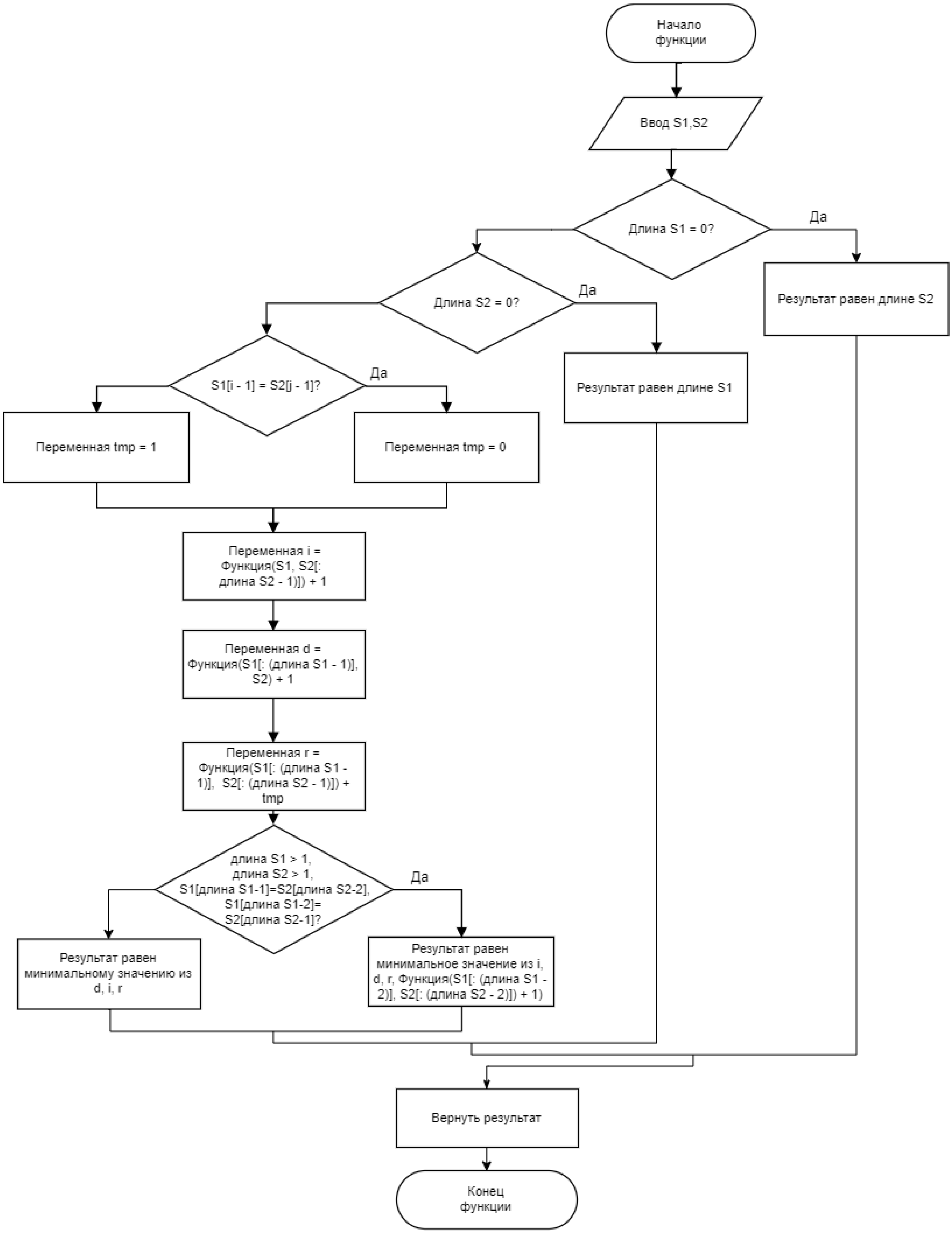

Рис. 2.5: Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна

2.2 Сравнительный анализ матричной и рекурсивной реализаций

Рекурсивная реализация работает медленнее по сравнению с матричной из-за повторных вычислений, возникающих в ходе работы рекурсивного алгоритма. Это наглядно видно на Рис.2.6, иллюстрирующем дерево рекурсивных вызовов. При каждом рекурсивном вызове необходимо передавать в функцию подстроки исходных строк, что затратно по памяти. Эту проблему возможно избежать, если занести строки в глобальные переменные, что является плохой практикой.

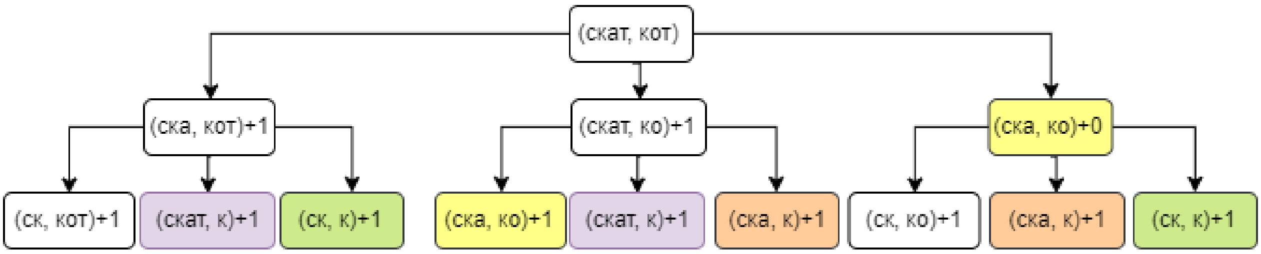

Рис. 2.6: Дерево рекурсивных вызовов

Для сравнения, вызов функции, которая реализует матричный алгоритм, происходит один раз и в функцию только один раз передаются обе строки. В этом алгоритме память будет затрачена на хранение матрицы, а время на вложенные циклы, однако затраты будут существенно меньше, чем при многократных вызовах рекурсивной функции.

Пусть длина строки S1 - n, длина строки S2 - m, тогда затраты памяти на приведенные выше алгоритмы будут следующими:

- матричный алгоритм Левенштейна:
  - строки S1, S2 - (m + n) * sizeof(char)
  - матрица - ((m + 1) * (n + 1)) * sizeof(int)
  - текущая строка матрицы - (n + 1) * sizeof(int)
  - длины строк - 2 * sizeof(int)
  - вспомогательные переменные - 3 * sizeof(int)
- матричный алгоритм Дамерау-Левенштейна:
  - строки S1, S2 - (m + n) * sizeof(char)
  - матрица - ((m + 1) * (n + 1)) * sizeof(int)
  - текущая строка матрицы - (n + 1) * sizeof(int)
  - длины строк - 2 * sizeof(int)
  - вспомогательные переменные - 3 * sizeof(int)

- рекурсивный алгоритм Дамерау-Левенштейна(для каждого вызова):
  - строки S1, S2 - (m + n) * sizeof(char)
  - длины строк - 2 * sizeof(int)
  - вспомогательная переменная - sizeof(int)
  - адрес возврата

3 Технологическая часть

В данном разделе приведены требования к программному обеспечению, средства реализации и листинг кода

3.1 Требования к ПО

Программа на вход получает две строки символов. Результат работы программы: число - искомое расстояние. Для матричных реализаций дополнительно выводится получившаяся в результате работы программы матрица.

3.2 Средства реализации

Для реализации представленных алгоритмов был выбран язык Python. Время работы алгоритмов было замерено с помощью функции time() из библиотеки time.

3.3 Листинги кода

В Листинге 3.1 показана реализация матричного алгоритма нахождения расстояния Левенштейна

Листинг 3.1: Функция нахождения расстояния Левенштейна матрично

```
def levenshtein_matrix(s1, s2, matrix_flag):
    n, m = len(s1), len(s2)
    current_row = range(n + 1)
    matrix = [current_row]
    for i in range(1, m + 1):
        current_row = [i] + [0] * n
        for j in range(1, n + 1):
            current_row[j] = matrix[i - 1][j - 1]
            if s1[j - 1] != s2[i - 1]:
                current_row[j] += 1
            current_row[j] = min(current_row[j], matrix[i-1][j] + 1,
                                 current_row[j - 1] + 1)
        matrix.append(current_row)
    if matrix_flag:
        print_matrix(matrix)
    return matrix[m][n]
```

В Листинге 3.2 показана реализация матричного алгоритма нахождения расстояния Дамерау-Левенштейна

Листинг 3.2: Функция нахождения расстояния Дамерау-Левенштейна матрично

```
def damerau_levenshtein_matrix(s1, s2, matrix_flag):
    n, m = len(s1), len(s2)
    current_row = range(n + 1)
    matrix = [current_row]
    for i in range(1, m + 1):
        current_row = [i] + [0] * n
        for j in range(1, n + 1):
            current_row[j] = matrix[i - 1][j - 1]
            if s1[j - 1] != s2[i - 1]:
                current_row[j] += 1
            current_row[j] = min(current_row[j], matrix[i-1][j] + 1,
                                 current_row[j - 1] + 1)
            if i > 1 and j > 1 and s1[j - 1] == s2[i - 2] \
                    and s1[j - 2] == s2[i - 1]:
                current_row[j] = min(current_row[j],
                                     matrix[i - 2][j - 2] + 1)
        matrix.append(current_row)
    if matrix_flag:
        print_matrix(matrix)
    return matrix[m][n]
```

В Листинге 3.3 показана реализация рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

Листинг 3.3: Функция нахождения расстояния Дамерау-Левенштейна рекурсивно

```
def damerau_levenshtein_recursive(s1, s2):
    n, m = len(s1), len(s2)
    if n == 0 or m == 0:
        if n != 0:
            return n
        if m != 0:
            return m
        return 0
    change = 0
    if s1[-1] != s2[-1]:
        change += 1
    if n > 1 and m > 1 and s1[-1] == s2[-2] and s1[-2] == s2[-1]:
        return min(damerau_levenshtein_recursive(s1[:n - 1], s2)
                   + 1,
                   damerau_levenshtein_recursive(s1, s2[:m - 1])
                   + 1,
                   damerau_levenshtein_recursive(s1[:n - 1],
                                                 s2[:m - 1])
                   + change,
                   damerau_levenshtein_recursive(s1[:n - 2],
```

```
                                                s2[:m - 2]) + 1)
    else:
        return min(damerau_levenshtein_recursive(s1[:n - 1], s2)
                   + 1,
                   damerau_levenshtein_recursive(s1, s2[:m - 1])
                   + 1,
                   damerau_levenshtein_recursive(s1[:n - 1],
                                                s2[:m - 1])
                   + change)
```

4 Эксперементальная часть

В данном разделе приведены примеры работы программы, постановка эксперимента и сравнительный анализ алгоритмов на основе экспериментальных данных.

4.1 Примеры работы

Пример 1
Строка s1 = кот
Строка s2 = скат
Расстояние Левенштейна(матричный алгоритм): 2
0 1 2 3
1 1 2 3
2 1 2 3
3 2 2 3
4 3 3 2
Расстояние Дамерау-Левенштейна(матричный алгоритм):2
0 1 2 3
1 1 2 3
2 1 2 3
3 2 2 3
4 3 3 2
Расстояние Дамерау-Левенштейна(рекурсивный алгоритм):2

Пример 2
Строка s1 = отчет
Строка s2 = спать
Расстояние Левенштейна(матричный алгоритм): 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 3 4 4 4
5 5 4 4 5 5
Расстояние Дамерау-Левенштейна(матричный алгоритм):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 3 4 4 4
5 5 4 4 5 5
Расстояние Дамерау-Левенштейна(рекурсивный алгоритм):5

Пример 3
Строка s1 = кофе
Строка s2 = коеф
Расстояние Левенштейна(матричный алгоритм): 2
0 1 2 3 4
1 0 1 2 3
2 1 0 1 2
3 2 1 1 1
4 3 2 1 2
Расстояние Дамерау-Левенштейна(матричный алгоритм):1
0 1 2 3 4
1 0 1 2 3
2 1 0 1 2
3 2 1 1 1
4 3 2 1 1
Расстояние Дамерау-Левенштейна(рекурсивный алгоритм):1

4.2 Функциональное тестирование

Было проведено функциональное тестирование программы, результаты которого занесены в Таблицу 4.1,1 столбец которой - номер тестового случая; 2 и 3 столбцы - строки, поступающие на вход; 4 столбец - ожидаемый результат, где

- 1-ая цифра - результат работы матричного алгоритма Левенштейна
- 2-ая цифра - результат работы матричного алгоритма Дамерау-Левенштейна
- 3-я цифра - результат работы рекурсивного алгоритма Дамерау-Левенштейна

5 столбец - полученный результат.

№	S1	S2	Ожидаемый результат	Полученный результат
1	пустая строка	пустая строка	0, 0, 0	0, 0, 0
2	кот	пустая строка	3, 3, 3	3, 3, 3
3	пустая строка	кот	3, 3, 3	3, 3, 3
4	кот	кот	0, 0, 0	0, 0, 0
5	кот	кит	1, 1, 1	1, 1, 1
6	от	кот	1, 1, 1	1, 1, 1
7	кот	кота	1, 1, 1	1, 1, 1
7	кот	кто	2, 1, 1	2, 1, 1

Таблица 4.1: Тестовые случаи

Программа успешно прошла все тестовые случаи, все полученные результаты совпали с ожидаемыми.

4.3 Сравение матричных алгоритмов

При сравнении быстродействия матричных алгоритмов были использованы строки длиной в диапазоне от 100 до 1000 с шагом 100. Количество повторов каждого эксперимента = 100. Результат одного эксперимента рассчитывается как средний из результатов проведенных испытаний с одинаковыми входными данными.

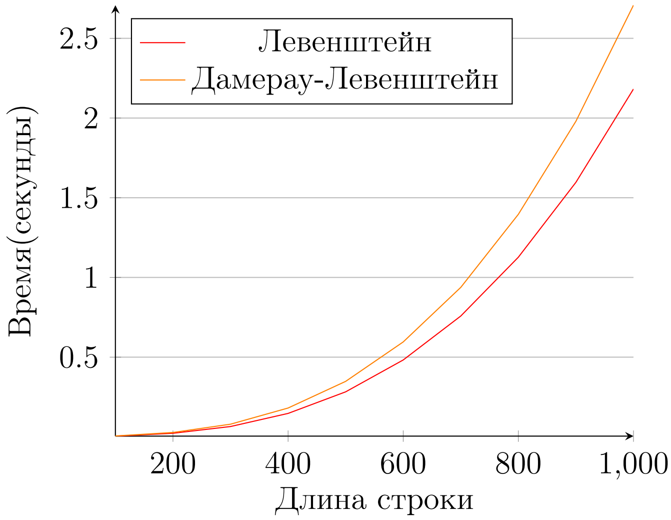

Рис. 4.1: Матричные алгоритмы Левенштейна и Дамерау-Левенштейна

На Рис. 4.1 видно, что матричный алгоритм Левенштейна превосходит матричный алгоритм Дамерау-Левенштейна примерно на 20%. Эта разница объясняется необходимостью дополнительных проверок на возможность транспозиции.

4.4 Сравнение реализаций алгоритма Дамерау-Левенштейна

При сравнении быстродействия матричного и рекурсивного алгоритмов Дамерау-Левенштейна были использованы строки длиной в диапазоне от 1 до 10 с шагом 1. Количество повторов каждого эксперимента = 100. Результат одного эксперимента рассчитывается как средний из результатов проведенных испытаний с одинаковыми входными данными.

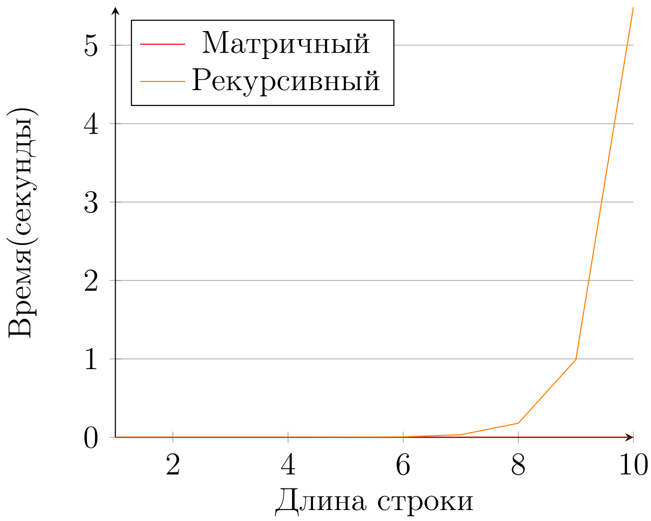

Рис. 4.2: Матричный и рекурсивный алгоритмы Дамерау-Левенштейна

На Рис. 4.1 видно, что матричный алгоритм Дамерау-Левенштейна начинает превосходить рекурсивный алгоритм Дамерау-Левенштейна при достижении длины строки 2, и превосходство увеличивается в геометрической прогрессии.

Заключение

В ходе работы были достигнуты все поставленные задачи. Были изучены и реализованы в матричной и рекурсивной форме алгоритмы Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками. Также был проведен сравнительный анализ матричной и рекурсивной реализаций алгоритма Левенштейна и матричных реализаций алгоритмов Левенштейна и Дамерау-Левенштейна по затрачиваемым ресурсам. Экспериментально подтверждено различие во временной эффективности рекурсивной и матричной реализаций путем замеров времени работы алгоритмов.