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 курсу "Анализ алгоритмов"

Тема Расстояния Левенштейна и Дамерау-Левенштейна

Студент Виноградов А. О.

Группа ИУ7-56Б

Оценка (баллы)

Преподаватели Волкова Л. Л., Строганов Ю. В.

Москва — 2022 г.

Содержание

Введение

Важной частью программирования является работа со строками. Часто возникает потребность в сравнении строк по длине и по содержимому. О таких алгоритмах и пойдет речь в данной работе.

Целью данной лабораторной работы является изучение, реализация и исследование алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна.

Задачи данной лабораторной работы:

1) изучение расстояний Левенштейна и Дамерау-Левенштейна;
2) разработка и реализация алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна;
3) проведение сравнительного анализа по времени матричной, рекурсивной и рекурсивной с использованием кэша реализаций алгоритма нахождения расстояния Дамерау-Левенштейна;
4) описание и обоснование полученных результатов в отчете о выполненной лабораторной работе.

1 Аналитическая часть

В данном разделе будут разобраны алгоритмы нахождения расстояний Левенштейна и Дамерау-Левенштейна.

1.1 Расстояние Левенштейна

Расстояние Левенштейна [1] между двумя строками — величина, определяемая как минимальное количество редакционных изменений, необходимых для преобразования одной строки в другую.
Выделяют следующие типы редакционных преобразований:

– I (Insert) — вставка символа в произвольной позиции;

– D (Delete) — удаление символа в произвольной позиции;

– R (Replace) — замена символа на другой;

M (Match) — совпадение двух символов (цена — 0).

Во время нахождения минимального количества редакционных изменений возникает проблема взаимного выравнивания строк. Рассмотрим пример для строк "развлечения"и "увлечение он представлен на рис. 1.1.

$$\left.\begin{array}{c|c|c|c|c|c|c|c|c|c|c}
\text{р} & \text{а} & \text{з} & \text{в} & \text{л} & \text{е} & \text{ч} & \text{е} & \text{н} & \text{и} & \text{я} \\
\text{у} & \text{в} & \text{л} & \text{е} & \text{ч} & \text{е} & \text{н} & \text{и} & \text{е} & & \\
\text{R} & \text{R} & \text{R} & \text{R} & \text{R} & \text{M} & \text{R} & \text{R} & \text{R} & \text{D} & \text{D} \\
1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1
\end{array}\right\}\text{10 редакционных изменений}$$

$$\left.\begin{array}{c|c|c|c|c|c|c|c|c|c|c}
\text{р} & \text{а} & \text{з} & \text{в} & \text{л} & \text{е} & \text{ч} & \text{е} & \text{н} & \text{и} & \text{я} \\
 & & \text{у} & \text{в} & \text{л} & \text{е} & \text{ч} & \text{е} & \text{н} & \text{и} & \text{е} \\
\text{D} & \text{D} & \text{R} & \text{M} & \text{M} & \text{M} & \text{M} & \text{M} & \text{M} & \text{M} & \text{R} \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}\right\}\text{4 редакционных изменения}$$

Таблица 1.1

Чтобы снять проблему выравнивания используют рекуррентную формулу (1.1). Пусть даны строки S_1 и S_2 длины N и M соответственно. Тогда $S_1[1 \ldots i]$ — подстрока S_1 длины i, начиная с первого символа. Аналогично, $S_2[1 \ldots j]$ — подстрока S_2 длины j, начиная с первого символа.

Расстояние Левенштейна между подстроками $S_1[1 \ldots i]$ и $S_2[1 \ldots j]$ находится по формуле 1.1

$$D(i,j) = \begin{cases} 0, & \text{i = 0, j = 0} \\ i, & \text{j = 0, i > 0} \\ j, & \text{i = 0, j > 0} \\ \min(& \\ \qquad D(i,j-1)+1, & \\ \qquad D(i-1,j)+1, & \\ \qquad D(i-1,j-1)+m(i,j) & \\ ), & \text{иначе} \end{cases} \tag{1.1}$$

где функция $m(i,j)$ определяется по формуле

$$m(i,j) = \begin{cases} 0, & S_1[i] = S_2[j] \\ 1, & \text{иначе} \end{cases} \tag{1.2}$$

Рекурсивный алгоритм реализует формулу (1.1). Реализация данной формулы может быть неэффективна по времени выполнения, особенно при больших N и M, так как множество промежуточных значений $D(i, j)$ вычисляется повторно.

Альтернативным решением является использование матрицы для хранения расстояний $D(i, j)$. Тогда алгоритм нахождения расстояния Левенштейна представляет собой построчное заполнение матрицы $A_{(N+1)(M+1)}$ значениями $D(i, j)$. Первая строка и первый столбец матрицы заполняются по формуле

$$A[i][j] = \begin{cases} j, & i = 0 \\ i, & j = 0 \end{cases} \tag{1.3}$$

Остальные строки и столбцы заполняются по формуле

$$A[i][j] = min \begin{cases} A[i-1][j] + 1 \\ A[i][j-1] + 1 \\ A[i-1][j-1] + m(i, j) \end{cases} \tag{1.4}$$

Результат вычисления расстояния Левенштейна будет находится в ячейке матрицы $A[N, M]$.

1.2 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна [2] между двумя строками, состоящими из конечного числа символов — это минимальное число редакционных изменений, необходимых для перевода одной строки в другую. Является модификацией расстояния Левенштейна — к списку редакционных изменений добавлена операция *транспозиции*, то есть перестановки двух соседних символов.

Расстояние Дамерау-Левенштейна может быть найдено по формуле

$$
D(i,j) = \begin{cases}
\max(i,j), & \text{если } \min(i,j) = 0, \\
\min(& \\
\qquad d_{a,b}(i, j-1) + 1, & \\
\qquad d_{a,b}(i-1, j) + 1, & \\
\qquad d_{a,b}(i-1, j-1) + m(i,j), & \text{иначе} \\
\qquad \left[\begin{array}{ll} d_{a,b}(i-2, j-2) + 1, & \text{если } i,j > 1; \\ & a[i] = b[j-1]; \\ & b[j] = a[i-1] \\ \infty, & \text{иначе} \end{array} \right. & \\
) &
\end{cases}, \tag{1.5}
$$

Формула опирается на те же соображения, что и формула (1.1), с учетом добавления операции транспозиции.

Рекурсивный алгоритм вычисления расстояния Дамерау-Левенштейна реализует формулу (1.5). Однако, как и в рекурсивном алгоритме нахождения расстояния Левенштейна, в данном алгоритме происходит многократное вычисление промежуточных $D(i,j)$. В качестве оптимизации можно использовать матрицу для хранения уже вычисленных промежуточных $D(i,j)$.

Матричный алгоритм нахождения расстояния Дамерау-Левенштейна представляет собой построчное заполнение матрицы $A_{(N+1)(M+1)}$ значениями $D(i,j)$. Результат вычисления расстояния Дамерау-Левенштейна будет находится в ячейке матрицы $A[N,M]$. Первая строка и первый столбец матрицы заполняются по формуле

$$
A[i][j] = \begin{cases} j, & i = 0 \\ i, & j = 0 \end{cases} \tag{1.6}
$$

Остальные строки и столбцы заполняются по формуле

$$
A[i][j] = min \begin{pmatrix} A[i-1][j] + 1, \\ A[i][j-1] + 1, \\ A[i-1][j-1] + m(i,j), \\ t(i,j) \end{pmatrix} \tag{1.7}
$$

где функция $t(i, j)$ определяется по формуле

$$t(i,j) = \left[\begin{array}{ll} A[i-2][j-2]+1, \text{ если } i>1, \ j>1, & S_1[i-2] = S_2[j-1], \\ & S_2[i-1] = S_2[j-2] \\ \infty, & \text{иначе} \end{array}\right. \tag{1.8}$$

Для сокращения количества вычислений промежуточных значений D(i, j) в рекурсивной реализации алгоритма, можно использовать кэш, представляющий собой матрицу. Если в матрице уже есть значение текущего D(i, j), это расстояние не рассчитывается снова, а берется из матрицы. Иначе происходит расчет текущего значения, которое заносится в матрицу.

Вывод

В данном разделе были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна. Формулы нахождения данных расстояний являются рекуррентными, что позволяет реализовывать алгоритмы как рекурсивно, так и итеративно.

2 Конструкторская часть

В этом разделе будут представлено описание используемых типов данных, а также схемы алгоритмов вычисления расстояния Левенштейна и Дамерау-Левенштейна.

2.1 Разработка алгоритмов

В данной части будут рассмотрены схемы алгоритмов нахождения расстояний Левештейна и Дамерау-Левенштейна. На рисунках 2.1 — 2.4 представлены схемы рассматриваемых алгоритмов.

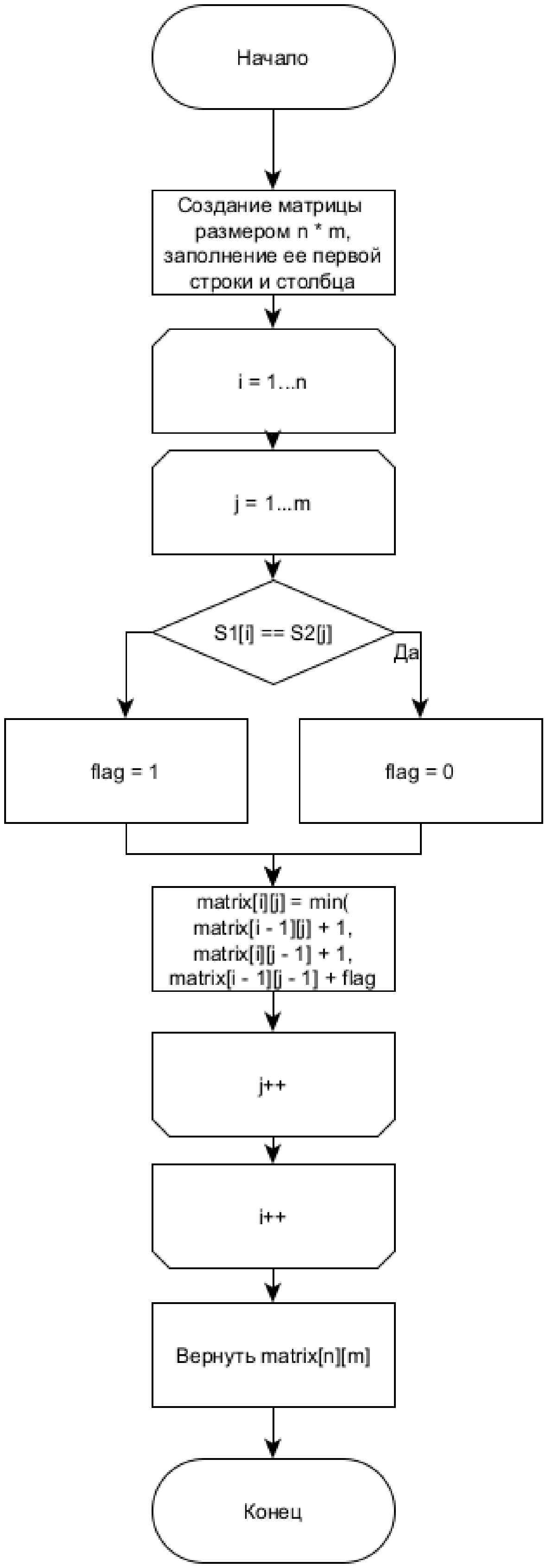

Рисунок 2.1 – Схема алгоритма нахождения расстояния Левенштейна

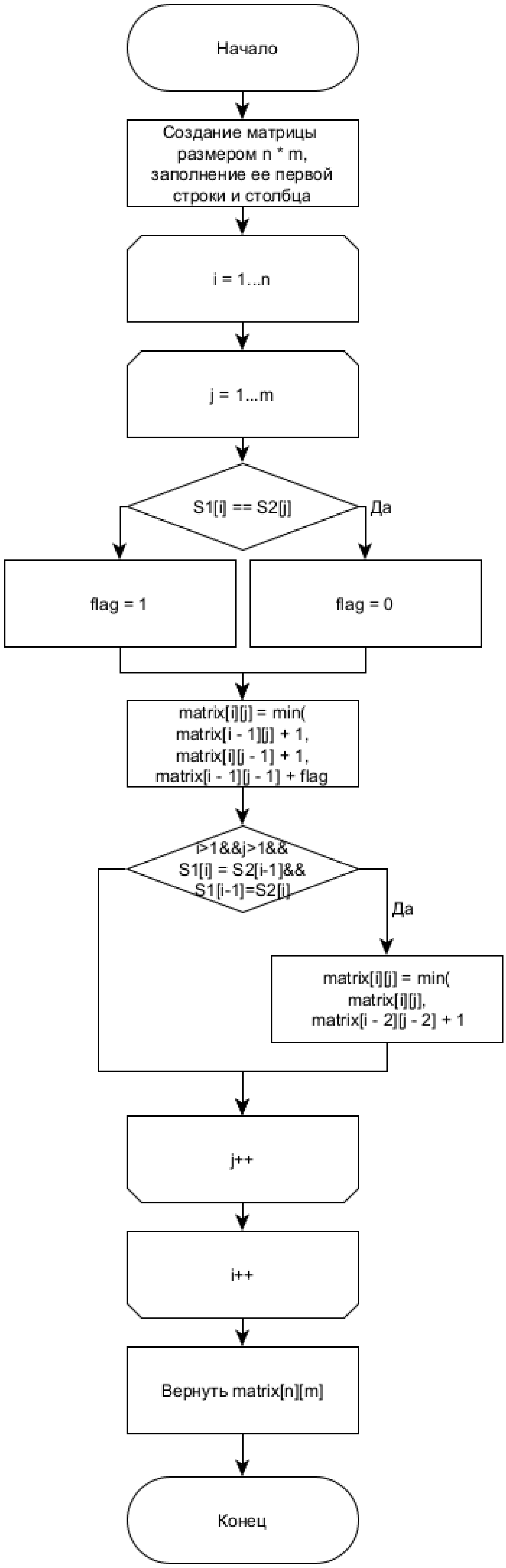

Рисунок 2.2 – Схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна

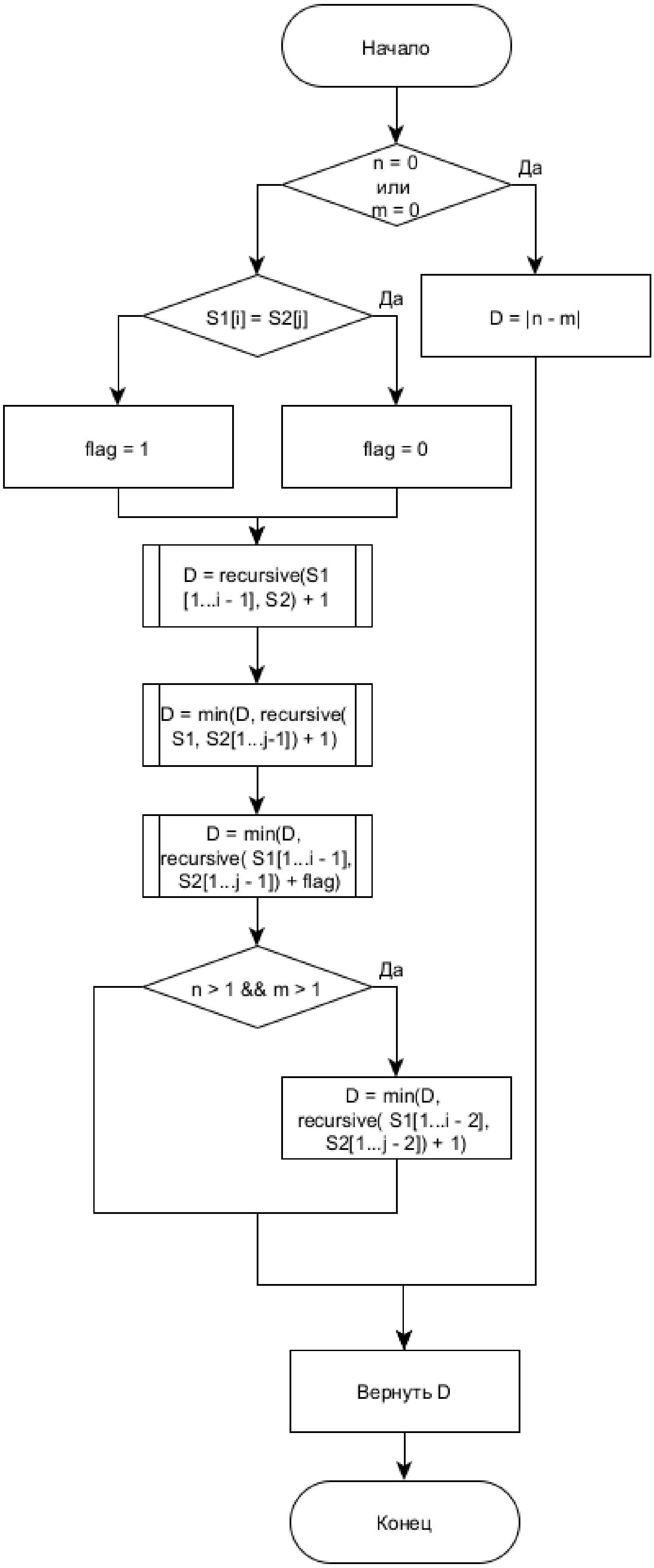

Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

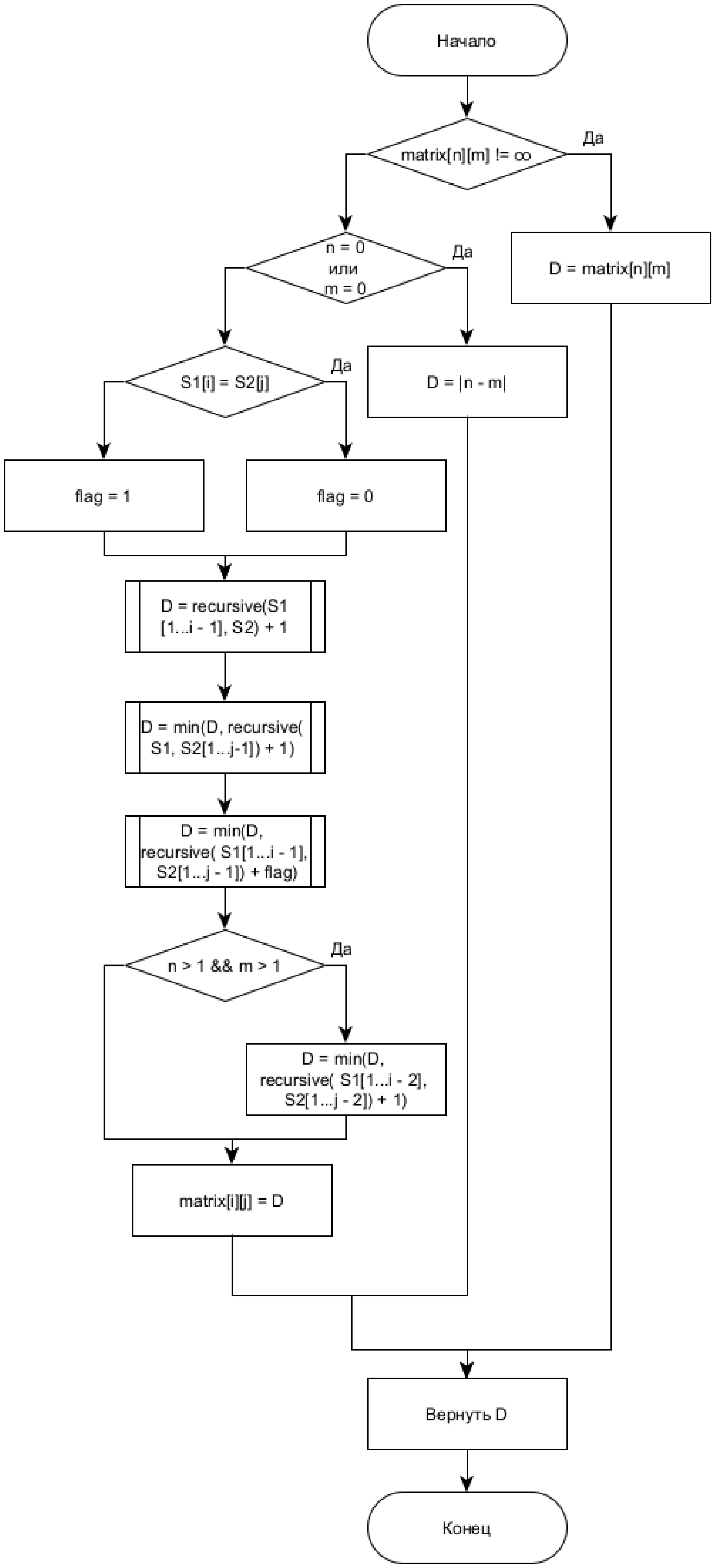

Рисунок 2.4 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с использованием кэша

Вывод

На основе теоретических данных, полученных в аналитическом разделе, были построены схемы исследуемых алгоритмов.

3 Технологическая часть

В данном разделе будут рассмотрены средства реализации, а также представлены листинги алгоритмов определения расстояния Левенштейна и Дамерау-Левенштейна.

3.1 Средства реализации

Для данной работы был выбран язык Python [3]. Для данной лабораторной работы требуются инструменты для работы со строками, замеров процессорного времени работы выполняемой программы, визуализации полученных данных. Все перечисленные инструменты присутствуют в выбранном языке программирования

3.2 Листинги кода

В листингах 3.1 — 3.5 приведена реализация алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

Листинг 3.1 – Функция нахождения расстояния Левенштейна итеративно

```
def levenshtein(s1:str, s2:str, n:int, m:int) -> int:
    if (n == 0):
        return m
    if (m == 0):
        return n

    matrix = [[0 for j in range(m + 1)] for i in range (n + 1)]
    for i in range(n + 1):
        matrix[i][0] = i
    for j in range(m + 1):
        matrix[0][j] = j

    cost = 0
```

Листинг 3.2 – Функция нахождения расстояния Левенштейна итеративно

```
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            cost = (s1[i - 1] != s2[j - 1])
            matrix[i][j] = min(matrix[i-1][j] + 1, matrix[i][j-1] +
               1, matrix[i-1][j-1] + cost)
    return matrix[n][m]
```

Листинг 3.3 – Функция нахождения расстояния Дамерау-Левенштейна итеративно

```
def damerau(s1:str, s2:str, n:int, m:int) -> int:
    if (n == 0):
        return m
    if (m == 0):
        return n

    matrix = [[0 for j in range(m + 1)] for i in range (n + 1)]
    for i in range(n + 1):
        matrix[i][0] = i
    for j in range(m + 1):
        matrix[0][j] = j

    cost = 0
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            cost = (s1[i - 1] != s2[j - 1])
            matrix[i][j] = min(matrix[i-1][j] + 1, matrix[i][j-1] +
               1, matrix[i-1][j-1] + cost)

            if (i > 1 and j > 1 and s1[i-1] == s2[j-2] and s1[i-2]
              == s2[j-1]):
                matrix[i][j] = min(matrix[i][j], matrix[i-2][j-2] +
                   1)

    return matrix[n][m]
```

Листинг 3.4 – Функция нахождения расстояния Дамерау-Левенштейна рекурсивно

```
def damerau_recursive(s1:str, s2:str, n:int, m:int) -> int:
    if (n * m == 0):
        return abs(n - m)

    res = min(damerau_recursive(s1, s2, n - 1, m),
        damerau_recursive(s1, s2, n, m-1)) + 1
    res = min(res, damerau_recursive(s1, s2, n - 1, m - 1) + (s1[n
        - 1] != s2[m - 1]))

    if (n > 1 and m > 1 and s1[n-1] == s2[m-2] and s1[n-2] ==
        s2[m-1]):
        res = min(res, damerau_recursive(s1, s2, n - 2, m - 2) + 1)

    return res
```

Листинг 3.5 – Функция нахождения расстояния Дамерау-Левенштейна рекурсивно с использованием кэша

```
def damer_rec_cache(s1:str, s2:str, n:int, m:int, cache:[]) -> int:
    if (n * m == 0):
        return abs(n - m)

    if (cache[n][m] > 0):
        return cache[n][m]

    res = min(damer_rec_cache(s1, s2, n - 1, m, cache),
        damer_rec_cache(s1, s2, n, m-1, cache)) + 1
    res = min(res, damer_rec_cache(s1, s2, n - 1, m - 1, cache) +
        (s1[n - 1] != s2[m - 1]))

    if (n > 1 and m > 1 and s1[n-1] == s2[m-2] and s1[n-2] ==
        s2[m-1]):
        res = min(res, damer_rec_cache(s1, s2, n - 2, m - 2, cache)
            + 1)
    cache[n][m] = res
    return res

def damerau_recursive_cache(s1:str, s2:str, n:int, m:int) -> int:
    cache = [[-1 for j in range(m + 1)] for i in range (n + 1)]
    return damer_rec_cache(s1, s2, n, m, cache)
```

3.3 Функциональные тесты

В таблице 3.1 приведены тесты для функций, реализующих алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна.

Таблица 3.1 – Функциональные тесты

			Ожидаемый результат	
№	Строка 1	Строка 2	Левенштейн	Дамерау-Левенштейн
1			0	0
2		teststr	7	7
3	smthfortest		11	11
4	test	nest	1	1
5	usage	sausage	2	2
6	phn	phone	2	2
7	ucnle	uncle	2	1
8	sure	user	3	2
9	plaanee	plane	2	2
10	labtpoe	laptop	3	3

Все тесты пройдены функциями успешно.

Вывод

В данном разделе были разработаны и приведены исходные коды четырех алгоритмов, рассмотренных и описанных ранее. Также, были описаны выбранные средства реализации алгоритмов и функциональные тесты для описанных алгоритмов.

4 Исследовательская часть

Демонстрация работы программы приведена на рисунке 4.1.

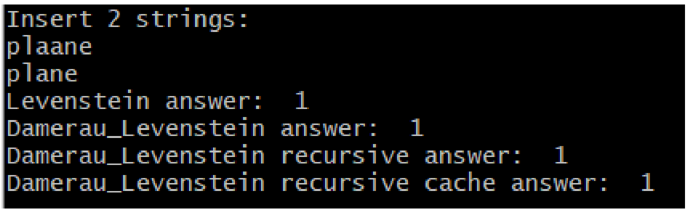

Рисунок 4.1 – Работа алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна.

4.1 Технические характеристики

Ниже приведены технические характеристики устройства, на котором было проведено измерение времени работы ПО:

- операционная система Windows 10 Домашняя Версия 21H1 [4] x86_64;
- оперативная память 8 Гб 2133 МГц;
- процессор Intel Core i5-8300H с тактовой частотой 2.30 ГГц [5].

4.2 Время выполнения реализаций алгоритмов

Для замеров времени используется функция замера процессорного времени process_time из библиотеки time на Python. Функция возвращает процессорное время типа float [6].

Функция используется дважды — в начале и в конце замера времени, затем значения начальное значение вычитается из конечного.

Замеры проводились для слов длины от 0 до 9 раз на строках равной длины по 1000 раз.

В таблице 4.1 представлены замеры времени работы для каждого из алгоритмов.

Таблица 4.1 – Результаты замеров времени (в мс)

Длина	Лев.(матр.)	Д.-Л.(матр.)	Д.-Л.(рек.)	Д.-Л.(рек. с кэшем)
0	0.0011	0.0005	0.0	0.0014
1	0.0027	0.0023	0.0	0.0028
2	0.005	0.0044	0.0	0.0058
3	0.0064	0.0073	0.0	0.0106
4	0.0109	0.0094	0.1562	0.0172
5	0.0125	0.0156	0.625	0.025
6	0.0172	0.0219	2.9688	0.0328
7	0.0234	0.025	16.875	0.0453
8	0.0281	0.0344	91.25	0.0578
9	0.0344	0.0422	503.125	0.075

На рисунках 4.2, 4.3, 4.4 приведены графические результаты замеров.

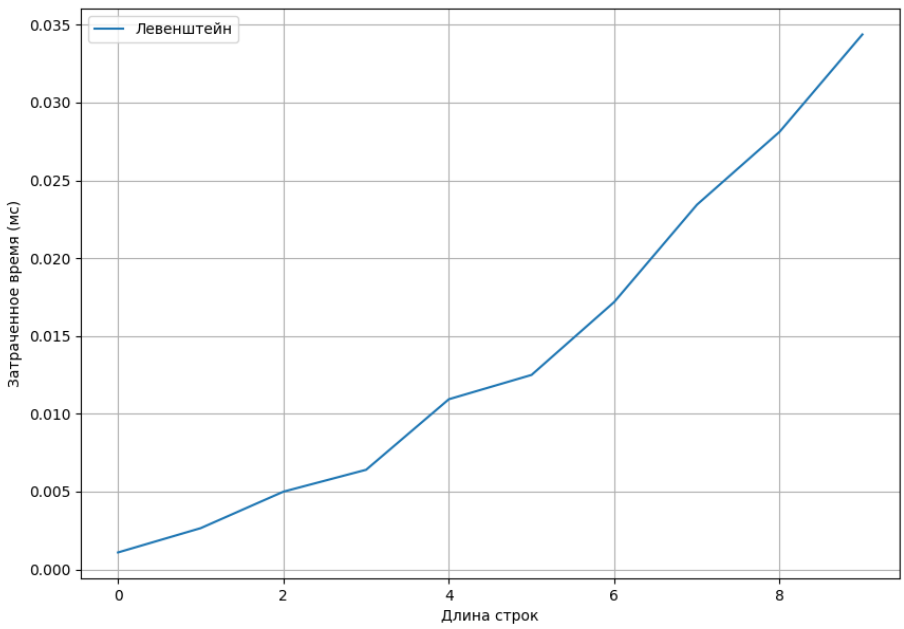

Рисунок 4.2 – Результаты работы алгоритма поиска расстояния Левенштейна для строк длины от 0 до 9

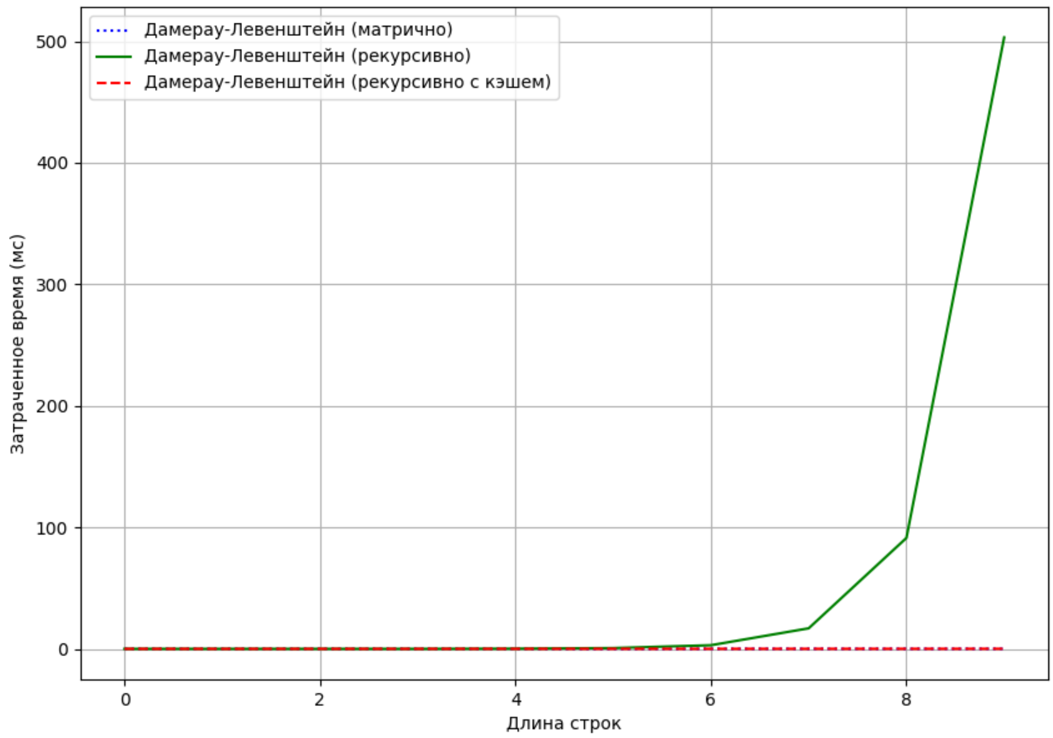

Рисунок 4.3 – Сравнение результатов работы реализаций алгоритмов поиска расстояния Дамерау-Левенштейна для строк длины от 0 до 9

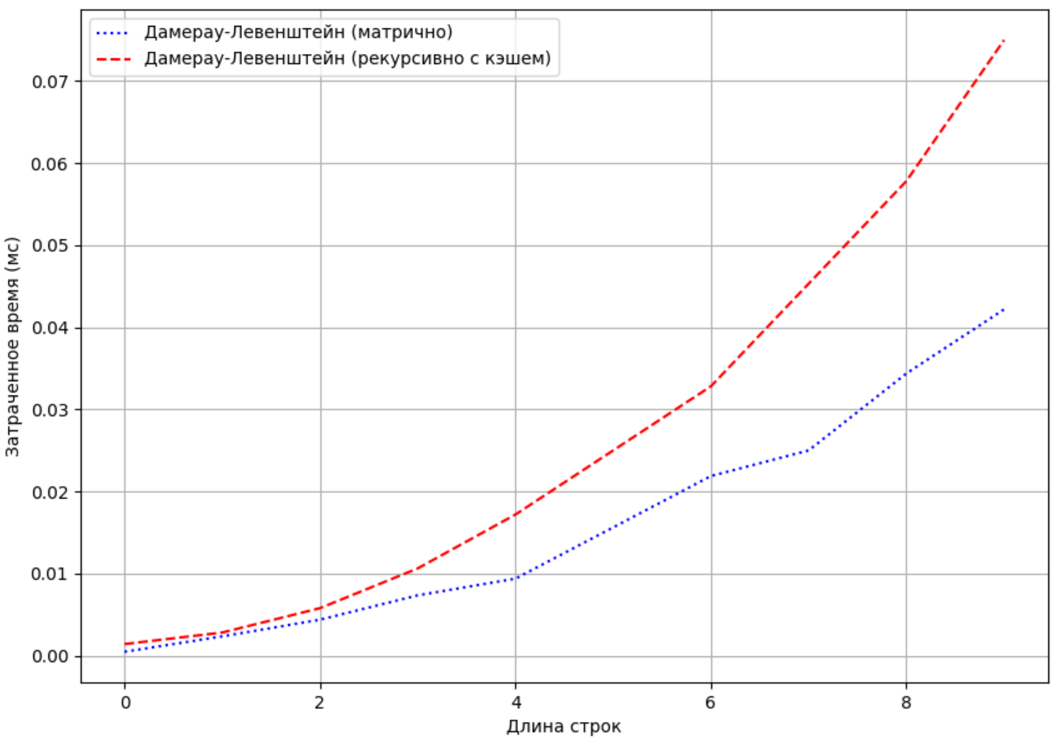

Рисунок 4.4 – Результаты работы реализаций алгоритмов поиска расстояния Левенштейна для строк длины от 0 до 9

Из полученных результатов можно сделать вывод, что алгоритм поиска расстояния Левенштейна в общем случае имеет сложность $O(N^2)$ (Рис. 4.2).

Рекурсивная реализация поиска расстояния Дамерау-Левенштейна сильно уступает по временным характеристикам матричной реализации и реализации с кэшем (Рис. 4.3).

4.3 Использование памяти

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

- Для матричного алгоритма нахождения расстояния Левенштейна(Дамерау-Левенштейна) расходы памяти составляют:
  - $sizeof(char) \cdot (n + m)$ — для двух строк длин n и m
  - $sizeof(int) \cdot ((n + 1) \cdot (m + 1))$ — для матрицы промежуточных значений
  - $sizeof(int) \cdot 2$ — для n и m
  - $sizeof(int) \cdot 2$ — вспомогательные локальные переменные
  - адрес возврата
- Единоразовые расходы памяти для рекурсивной реализации:
  - $sizeof(char) \cdot (n + m)$ — для двух строк длин n и m

  Расходы памяти для каждого вызова функции рекурсивной реализации (максимум n + m вызовов):

  - $sizeof(int) \cdot 2$ — для n и m
  - $sizeof(int) \cdot 1$ — вспомогательная локальная переменная
  - адрес возврата

  Максимальные затраты памяти рекурсивного алгоритма достигаются при максимальной глубине рекурсии, равной сумме длин строк(n + m). Таким образом, теоретические максимальные затраты памяти рекурсивной реализации составляют $(sizeof(char) \cdot (n+m) + sizeof(int) \cdot 3) \cdot (n + m)$

– Также для рекурсивного алгоритма с кэшем следует учитывать единичные (не для каждого шага рекурсии, а только единожды) затраты памяти на хранение матрицы-кэша $((n+1) \cdot (m+1)) \cdot sizeof(int))$

Вывод

В результате экспериментов было выявлено, что обычный матричный алгоритм поиска расстояния Дамерау-Левенштейна сравним по затратам ресурсов с рекурсивной реализацией с использованием кэша, однако следует учитывать дополнительные затраты времени и памяти на многочисленные рекурсивные вызовы, делающие матричный алгоритм более эффективным по ресурсам.

Обычная рекурсивная реализация (без кэша) выигрывает у матричного метода по затратам памяти, т. к. максимальные затраты памяти рекурсивного варианта растут как функция от $(n+m)$, а у матричного варианта — $(n+1) \cdot (m+1)$. Однако время выполнения рекурсивного варианта быстро растет и начинает многократно превосходить время выполнения матричной реализации уже начиная с длины строк равной 5.

Заключение

В результате исследования было определено, что время выполнения рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна на длинах строке больше 5 многократно (более чем в 6 раз) превосходит время выполнения матричной реализации алгоритма и рекурсивного алгоритма с кэшем. Однако скорость роста затрат памяти рекурсивного алгоритма линейна, в то время как затраты памяти других двух алгоритмов растут как функция $f(n \cdot m)$.

В результате исследования были получены следующие результаты замеров времени работы алгоритмов (для $n = m = 9$):

– 34 мс — поиск расстояния Левенштейна;

– 42 мс — матричная реализация поиска расстояния Дамерау-Левенштейна;

– 503 мс — рекурсивная реализация поиска расстояния Дамерау-Левенштейна;

– 75 мс — рекурсивная реализация поиска расстояния Дамерау-Левенштейна с кэшем.

Для строк длины 9 время выполнения составляет 42 мс для матричного метода и 503 мс для рекурсивной функции (превосходство более чем в 11 раз)

Цель, поставленная в начале работы, была достигнута. Кроме того были достигнуты поставленные задачи:

– были изучены расстояния Левенштейна и Дамерау-Левенштейна;

– были разработаны и реализованы алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна;

- был проведен сравнительный анализ по времени матричной, рекурсивной и рекурсивной с использованием кэша реализаций алгоритма нахождения расстояния Дамерау-Левенштейна;
- был подготовлен отчет о лабораторной работе.

Список использованных источников

[1] Левенштейн В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов. — М.: издательство «Наука» Доклады АН СССР. Т. 163.

[2] Черненький В. М. Гапанюк Ю. Е. Методика идентификации пассажира по установочным данным. — М.: издательство МГТУ им. Н.Э. Баумана Вестник МГТУ им. Н.Э. Баумана. Сер. «Приборостроение», 2012. Т. 163. С. — 30–34.

[3] Welcome to Python [Электронный ресурс]. Режим доступа: `https://www.python.org` (дата обращения: 10.10.2022).

[4] Windows [Электронный ресурс]. Режим доступа: `https://www.microsoft.com/en-us/windows` (дата обращения: 01.10.2022).

[5] Процессор Intel Core i5 [Электронный ресурс]. Режим доступа: `https://www.intel.com/processors/core/i5/docs` (дата обращения: 01.10.2022).

[6] time — Time access and conversions [Электронный ресурс]. Режим доступа: `https://docs.python.org/3/library/time.html` (дата обращения: 10.10.2022).