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»

Отчёт по лабораторной работе № 1 по курсу «Анализ алгоритмов»

Тема Расстояние Левенштейна и Дамерау — Левенштейна

Студент Гаврилюк В. А.

Группа ИУ7-51Б

Оценка (баллы)

Преподаватель Волкова Л. Л.

Москва — 2024 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Расстояние Левенштейна (редакционное расстояние) между двумя строками — минимальное число необходимых редакционных операций — вставок, удалений и замен, необходимых для преобразования первой строки во вторую. При этом совпадения символов не являются операциями, которые учитываются в редакционном расстоянии, поэтому, если строки совпадают, то их редакционное расстояние равно нулю для любой длины строк [1].

Расстояние Дамерау — Левенштейна между двумя строками — минимальное число редакционных операций — вставок, удалений, замен и транспозиций соседних символов, необходимых для преобразования первой строки во вторую. Таким образом, для расстояния Дамерау — Левенштейна кроме основных операций вставки, удаления и замены вводится дополнительная операция транспозиции (transposition) двух соседних символов.

Расстояния Левенштейна и Дамерау — Левенштейна используются в ряде практически важных задач: проверка правописания, задачи поиска в текстовых базах данных, задачи исследования ДНК и т. д. [1].

Цель лабораторной работы — исследование алгоритмов нахождения расстояния Левенштейна и Дамерау — Левенштейна. Для достижения поставленной цели необходимо выполнить следующие задачи:

- построить схемы для рекурсивного алгоритма нахождения расстояния Левенштейна, рекурсивного алгоритма нахождения расстояния Левенштейна с мемоизацией, нерекурсивных алгоритмов нахождения расстояний Левенштейна и Дамерау — Левенштейна, основанных на идее динамического программирования;
- создать программное обеспечение (ПО), реализующее перечисленные выше алгоритмы;
- провести анализ потребляемых ресурсов (процессорное время и память) для перечисленных алгоритмов;
- описать и обосновать полученные результаты в отчёте.

1 Аналитический раздел

1.1 Расстояние Левенштейна

За редакционное расстояние принимают минимальное количество операций над строкой, с помощью которых она может быть преобразована в другую. Для каждой операции над строкой вводится условная цена. Суммарная стоимость всех произведённых операций и будет являться редакционным расстоянием между данными строками. Определим стоимости каждой редакционной операции:

- $\omega(\alpha, \beta) = 1$ — стоимость замены при $\alpha \neq \beta$;
- $\omega(\alpha, \alpha) = 0$ — стоимость эквивалентной замены;
- $\omega(\varnothing, \beta) = 1$ — стоимость вставки;
- $\omega(\alpha, \varnothing) = 1$ — стоимость удаления.

Пусть исходные строки S_1 и S_2 заданы посимвольно массивами $S_1[1...n]$ и $S_2[1...m]$. Пусть $D(i, j)$ — функция, значением которой является редакционное расстояние между подстроками $S_1[1...i]$ и $S_2[1...j]$. Функция $D(i, j)$ определяет минимальное число редакционных операций для преобразования первых i символов строки S_1 в первые j символов строки S_2. Таким образом, редакционное расстояние между строками S_1 и S_2 в точности равно $D(n, m)$. Для описания алгоритма вычисления расстояния Левенштейна используется рекуррентная формула (1.1).

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ D(i-1, j-1), & i > 0, j > 0, S_1[i] = S_2[j] \\ 1 + \min(& \\ \quad D(i, j-1), & \\ \quad D(i-1, j), & i > 0, j > 0, S_1[i] \neq S_2[j] \\ \quad D(i-1, j-1) & \\ ). & \end{cases} \tag{1.1}$$

1.2 Расстояние Дамерау — Левенштейна

Так как для вычисления расстояния Дамерау — Левенштейна вводится дополнительная операция транспозиции, формула (1.1) примет вид:

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ D(i-1, j-1), & i > 0, j > 0, S_1[i] = S_2[j] \\ 1 + min \begin{cases} D(i, j-1), \\ D(i-1, j), \\ D(i-1, j-1), \\ D(i-2, j-2), \end{cases} & \text{если } i > 1, j > 1, \\ & S_1[i] = S_2[j-1], \\ & S_1[i-1] = S_2[j], \\ 1 + min \begin{cases} D(i, j-1), \\ D(i-1, j), \\ D(i-1, j-1), \end{cases} & \text{иначе.} \end{cases} \tag{1.2}$$

1.3 Рекурсивный алгоритм Левенштейна

Простой рекурсивный алгоритм нахождения расстояния Левенштейна реализует рекуррентную формулу (1.1) на прямую. Но данный рекурсивный подход «сверху-вниз» (от англ. *top-down*) к вычислению $D(n, m)$ крайне неэффективен при больших значениях n и m. Проблема в том, что количество рекурсивных вызовов растёт экспоненциально с увеличением n и m. Но существует только $(n + 1) \times (m + 1)$ комбинаций i и j, поэтому возможно только $(n + 1) \times (m + 1)$ различных рекурсивных вызовов. Следовательно, неэффективность подхода «сверху-вниз» обусловлена огромным количеством избыточных рекурсивных вызовов процедуры [2].

1.4 Рекурсивный алгоритм Левенштейна с кэшем

Для того, чтобы исключить повторные вычисления, в рекурсивном алгоритме нахождения расстояния Левенштейна с кэшем используется мемоизация — способ оптимизации, смысл которого заключается в сохранении результата предыдущих вызовов функции $D(i, j)$ для некоторых i и j. Тогда при каждом вызове $D(i, j)$ будет проверяться, вызывалась ли функция для заданных i и j:

- если не вызывалась, то функция $D(i, j)$ вызывается, и результат её выполнения сохраняется;
- если вызывалась, то используется сохранённый результат.

1.5 Матричный алгоритм Левенштейна

Другой вариант нахождения расстояния Левенштейна — использование подхода «снизу-вверх» (от англ. *bottom-up*), который делится на два этапа:

- вычисление $D(i, j)$ для наименьших возможных значений i и j;
- вычисление $D(i, j)$ для остальных возрастающих значений i и j [2].

Обычно такой подход реализован с помощью матрицы результатов M размером $(n + 1) \times (m + 1)$, которая содержит значения $D(i, j)$ для всех пар i и j.

Значение в ячейке $M[i][j]$ соответствует значению $D(i, j)$. При вычислении значения для конкретной ячейки используются только ячейки

Таблица 1.1 – Матрица результатов, которая будет использоваться для расчёта редакционного расстояния между словами *vintner* и *writers*.

$D(i,j)$			**w**	**r**	**i**	**t**	**e**	**r**	**s**
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
v	1	1							
i	2	2							
n	3	3							
t	4	4							
n	5	5							
e	6	6							
r	7	7							

$M(i-1,j-1)$, $M(i,j-1)$ и $M(i-1,j)$, а также два символа $S_1(i)$ и $S_2(j)$. Таблица заполняется в соответствии с рекуррентным соотношением 1.1.

Стоит отметить, что для вычисления следующего элемента $D(i,j)$ нет необходимости хранить всю матрицу значений, т. к. используются только текущая и предыдущая строки. Поэтому для оптимизации использования памяти можно хранить только 2 строки.

1.6 Матричный алгоритм Дамерау — Левенштейна

Для матричного алгоритма нахождения расстояния Дамерау — Левенштейна все рассуждения аналогичны рассуждениям в разделе 1.5, но используется рекуррентная формула (1.2).

2 Конструкторский раздел

В данном разделе будут приведены требования к входным, выходным параметрам и представлены схемы для:

– рекурсивного алгоритма нахождения расстояния Левенштейна;

– рекурсивного алгоритма нахождения расстояния Левенштейна с мемоизацией;

– матричного алгоритма нахождения расстояния Левенштейна;

– матричного алгоритма нахождения расстояния Дамерау — Левенштейна.

2.1 Требования к входным и выходным параметрам

Требования к входным и выходным параметрам:

– в качестве входных параметров программа принимает две строки;

– символы разных регистров считаются различными;

– пустая строка является корректным входным значением;

– выходным параметром является число — расстояние Левенштейна или Дамерау — Левенштейна в зависимости от выбранного алгоритма.

2.2 Рекурсивный алгоритм Левенштейна

На рисунке 2.1 представлена схема рекурсивного алгоритма нахождения расстояния Левенштейна.

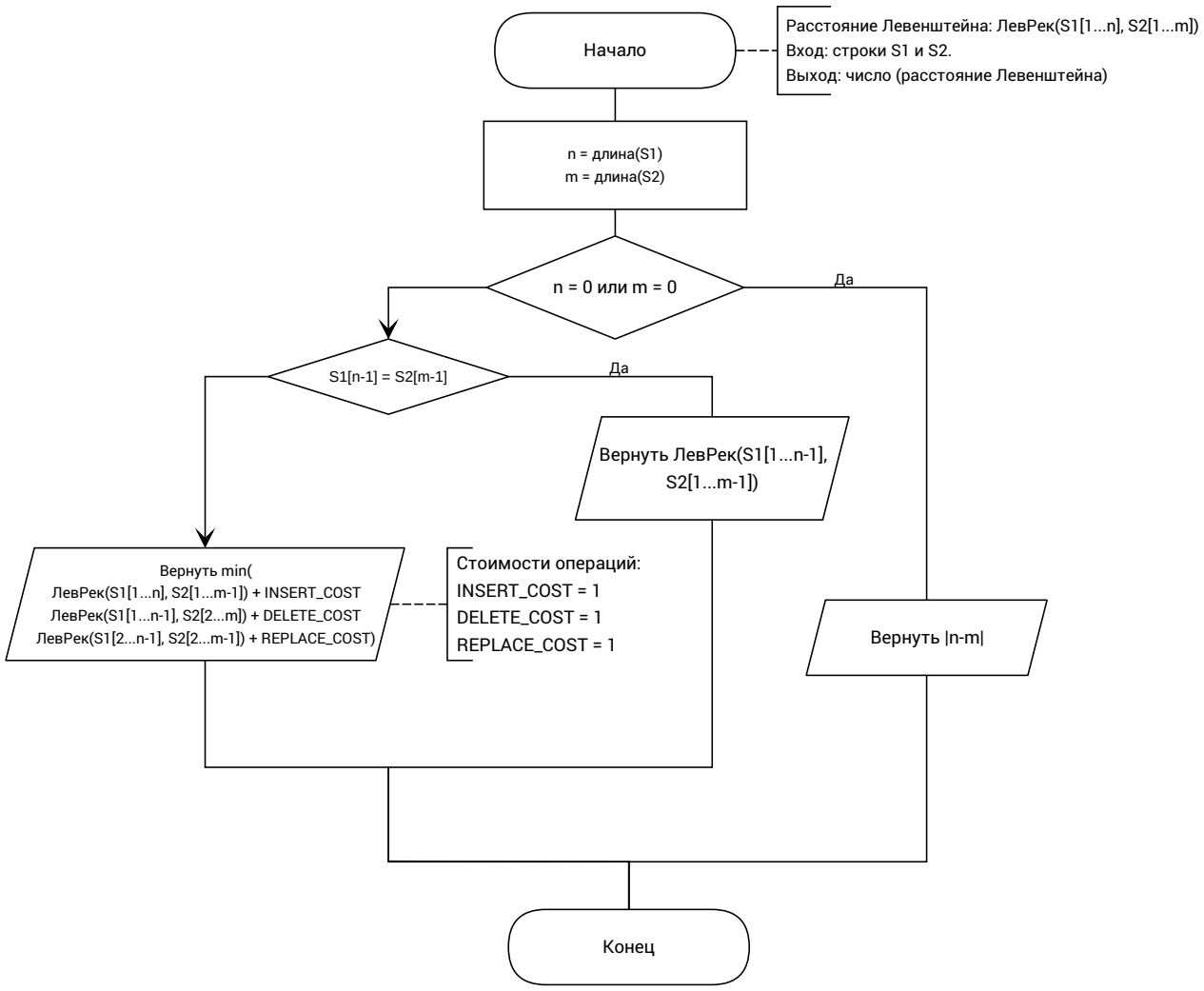


Рисунок 2.1 – Схема рекурсивного алгоритма нахождения расстояния Левенштейна

2.3 Рекурсивный алгоритм Левенштейна с кэшем

На рисунке 2.2 представлена схема рекурсивного алгоритма нахождения расстояния Левенштейна с мемоизацией.

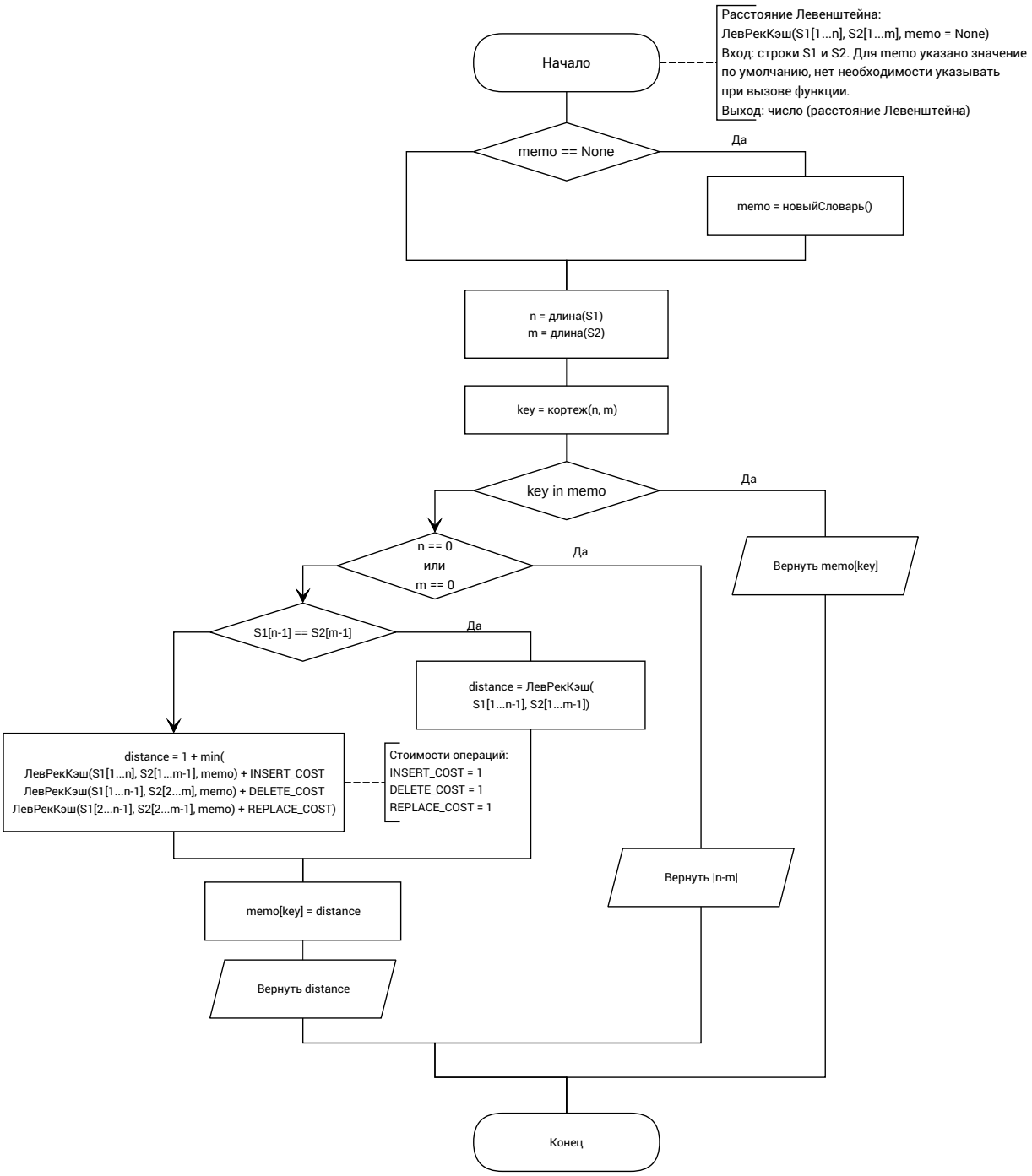


Рисунок 2.2 – Схема рекурсивного алгоритма нахождения расстояния Левенштейна с мемоизацией

2.4 Матричный алгоритм Левенштейна

На рисунке 2.3 представлена схема матричного алгоритма нахождения расстояния Левенштейна.

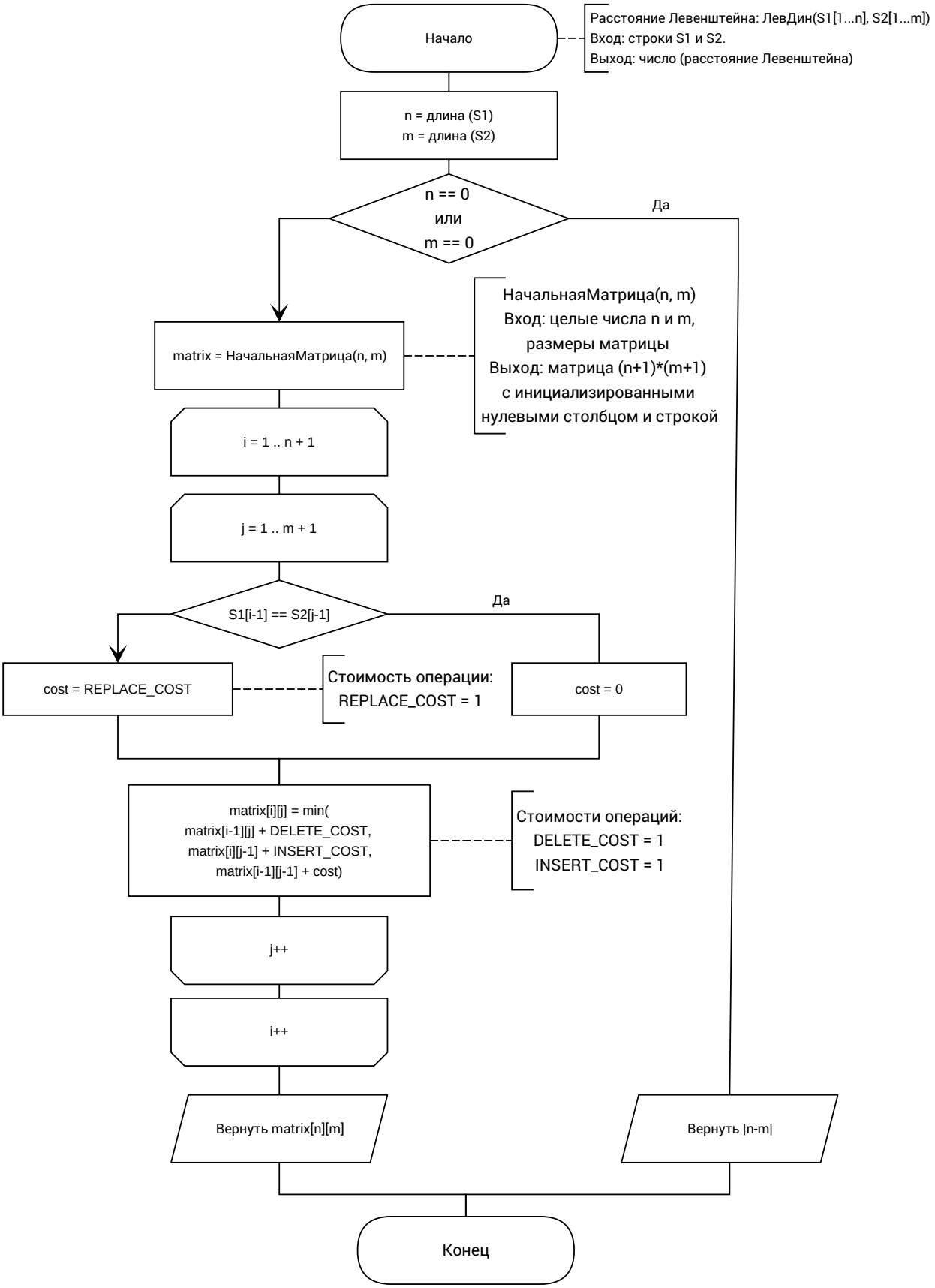


Рисунок 2.3 – Схема матричного алгоритма нахождения расстояния Левенштейна

2.5 Матричный алгоритм Дамерау — Левенштейна

На рисунке 2.4 представлена схема матричного алгоритма нахождения расстояния Дамерау — Левенштейна.

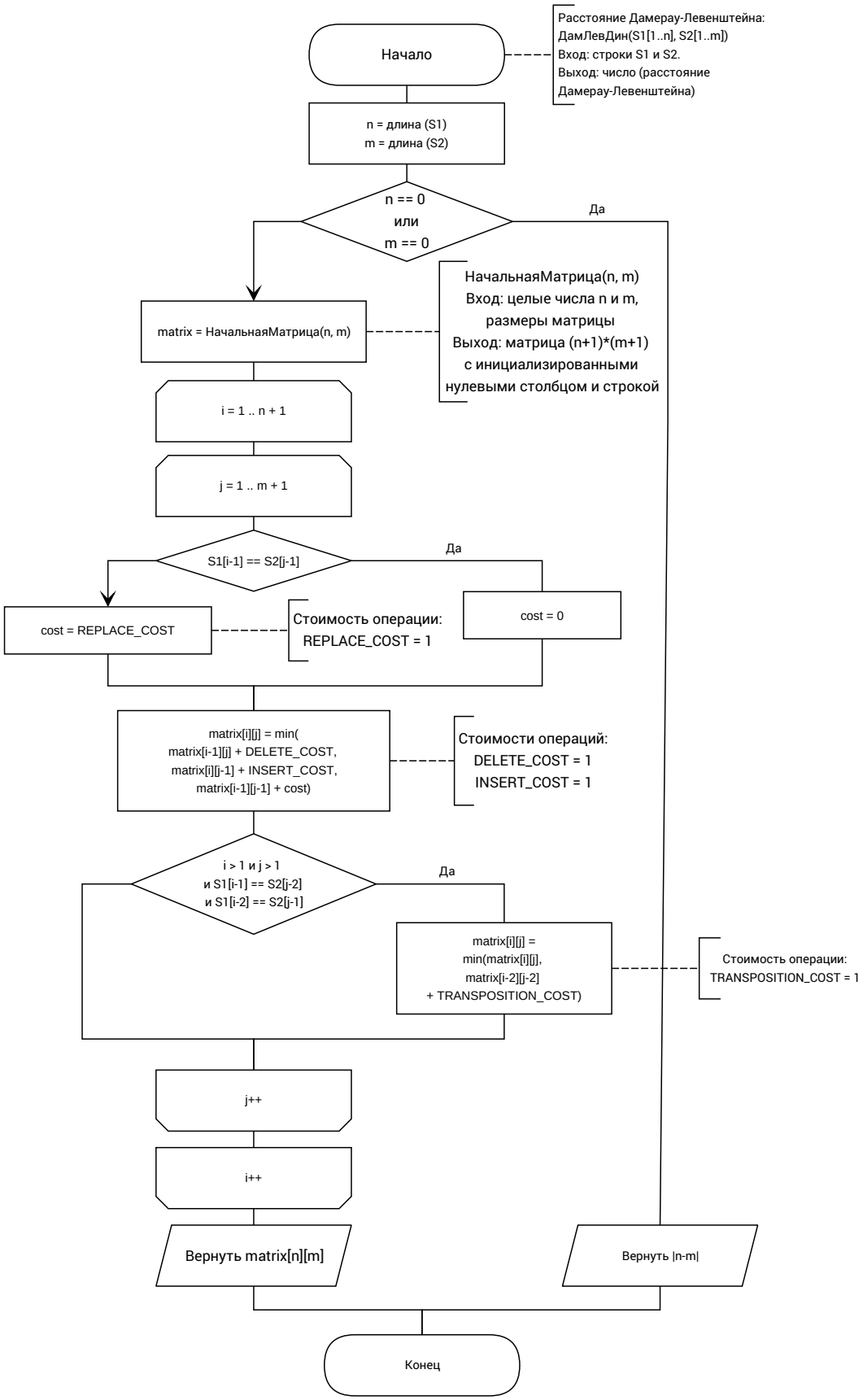


Рисунок 2.4 – Схема матричного алгоритма нахождения расстояния Дамерау — Левенштейна

3 Технологический раздел

В данном разделе будет представлена реализация алгоритмов поиска редакционного расстояния. Также будут указаны средства реализации алгоритмов и результаты тестирования.

3.1 Средства реализации

Для реализации был выбран язык программирования MicroPython [3]. Выбор обусловлен наличием функции вычисления процессорного времени в библиотеке utime [4]. Время было замерено с помощью функции time.ticks_ms().

3.2 Реализация алгоритмов

В листинге 3.1 представлено определение стоимости операций в виде глобальных переменных. Далее они будут использоваться в реализациях алгоритмов.

Листинг 3.1 – Определение стоимости операций

```
DELETE_COST = 1
INSERT_COST = 1
REPLACE_COST = 1
TRANSPOSITION_COST = 1
```

В листинге 3.2 представлены функции для создания матрицы и заполнения нулевого столбца и нулевой строки расстояниями, соответствующими преобразованиям из пустой первой строки S_1 в подстроки $S_2[1...i]$, где $i = \overline{1, m}$, и наоборот. Данные функции будут использоваться в матричных реализациях алгоритмов поиска редакционного расстояния.

Листинг 3.2 – Определение функций для инициализации матрицы для матричных алгоритмов

```
def createMatrix(rows: int, cols: int) -> list[list[int]]:
    return [[0 for _ in range(cols)] for _ in range(rows)]

def getInitialMatrix(rows: int, cols: int) -> list[list[int]]:
    matrix = createMatrix(rows+1, cols+1)
    for i in range(1, rows + 1):
        matrix[i][0] = matrix[i - 1][0] + DELETE_COST
    for j in range(1, cols + 1):
        matrix[0][j] = matrix[0][j - 1] + INSERT_COST

    return matrix
```

В листингах 3.3 — 3.6 представлены различные реализации алгоритмов нахождения редакционного расстояния Левенштейна и Дамерау — Левенштейна.

Листинг 3.3 – Реализация рекурсивного алгоритма поиска расстояния Левенштейна

```
def RecursiveLevenshtein(s1: str, s2: str) -> int:
    length1, length2 = len(s1), len(s2)
    if length1 == 0 or length2 == 0:
        return abs(length1 - length2)
    if s1[-1] == s2[-1]:
        return RecursiveLevenshtein(s1[:-1], s2[:-1])

    return min(
        RecursiveLevenshtein(s1, s2[:-1]) + INSERT_COST,
        RecursiveLevenshtein(s1[:-1], s2) + DELETE_COST,
        RecursiveLevenshtein(s1[:-1], s2[:-1]) + REPLACE_COST)
```

Листинг 3.4 – Реализация рекурсивного алгоритма поиска расстояния Левенштейна с мемоизацией

```
def RecursiveCacheLevenshtein(s1: str, s2: str, memo: dict =
   None) -> int:
    if memo is None:
        memo = {}

    length1, length2 = len(s1), len(s2)
    key = (length1, length2)
    if key in memo:
        return memo[key]

    if length1 == 0 or length2 == 0:
        return abs(length1 - length2)
    if s1[-1] == s2[-1]:
        distance = RecursiveCacheLevenshtein(s1[:-1], s2[:-1])
    else:
        distance = min(
            RecursiveCacheLevenshtein(s1, s2[:-1], memo) +
               INSERT_COST,
            RecursiveCacheLevenshtein(s1[:-1], s2, memo) +
               DELETE_COST,
            RecursiveCacheLevenshtein(s1[:-1], s2[:-1], memo) +
               REPLACE_COST)

    memo[key] = distance
    return distance
```

Листинг 3.5 – Реализация матричного алгоритма поиска расстояния Левенштейна

```
def DynamicLevenshtein(s1: str, s2: str) -> int:
    length1, length2 = len(s1), len(s2)
    if length1 == 0 or length2 == 0:
        return abs(length1 - length2)
    matrix = getInitialMatrix(length1, length2)

    for i in range(1, length1 + 1):
        for j in range(1, length2 + 1):
            cost = 0 if s1[i - 1] == s2[j - 1] else REPLACE_COST
            matrix[i][j] = min(
                matrix[i - 1][j] + DELETE_COST,
                matrix[i][j - 1] + INSERT_COST,
                matrix[i - 1][j - 1] + cost)

    return matrix[length1][length2]
```

Листинг 3.6 – Реализация матричного алгоритма поиска расстояния Дамерау — Левенштейна

```
def DynamicDamerauLevenshtein(s1: str, s2: str) -> int:
    length1, length2 = len(s1), len(s2)
    if length1 == 0 or length2 == 0:
        return abs(length1 - length2)
    matrix = getInitialMatrix(length1, length2)

    for i in range(1, length1 + 1):
        for j in range(1, length2 + 1):
            cost = 0 if s1[i - 1] == s2[j - 1] else REPLACE_COST
            matrix[i][j] = min(
                matrix[i - 1][j] + DELETE_COST,
                matrix[i][j - 1] + INSERT_COST,
                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] + TRANSPOSITION_COST)

    return matrix[length1][length2]
```

3.3 Тестирование

В таблице 3.1 представлены тесты для алгоритмов нахождения расстояний Левенштейна и Дамерау — Левенштейна. Тестирование проводилось по методологии чёрного ящика. Все тесты пройдены успешно.

Таблица 3.1 – Тесты для алгоритмов нахождения расстояний Левенштейна и Дамерау — Левенштейна

№	S_1	S_2	Ожидаемый результат	
			Левенштейн	Дамерау — Левенштейн
1	∅	∅	0	0
2	∅	hello	5	5
3	∅	привет	6	6
4	beauty	∅	6	6
5	невероятно	∅	10	10
6	abc	aba	1	1
7	кошка	мышка	2	2
8	aba	aab	2	1
9	котик	котки	2	1
10	abaa	aba	1	1
13	Hello world	eHllo kord	4	3

4 Исследовательский раздел

4.1 Технические характеристики

Замер процессорного времени был произведён на отладочной плате NUCLEO-F767ZI[5], позволяющей запускать программы, написанные на языке *MicroPython*[3].

Технические характеристики NUCLEO-F767ZI:

– разрядность шины данных — 32 бита;

– процессор — Arm Cortex-M7;

– объем ОЗУ — 512 Кбайт;

4.2 Измерение процессорного времени

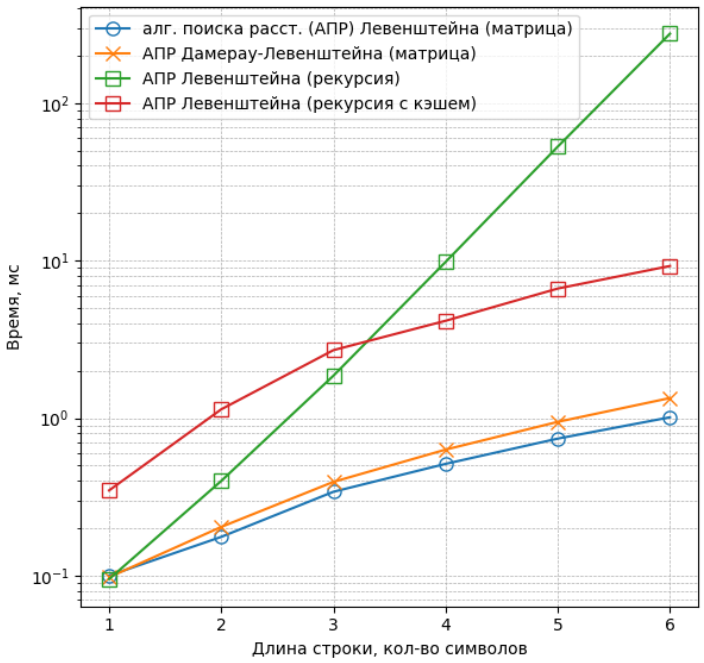
Измерение процессорного времени производилось для строк, состоящих из случайных последовательностей символов длиной от одного до шести в кодировке $UTF-8$. При этом длина входных строк в рамках замера совпадает. Результатом для i-ой ($i = \overline{1,6}$) длины является среднее арифметическое показателей $n = 15$ замеров.

В таблице 4.1 приведены результаты замеров процессорного времени работы алгоритмов нахождения расстояния Левенштейна (Л), и Дамерау — Левенштейна (ДЛ).

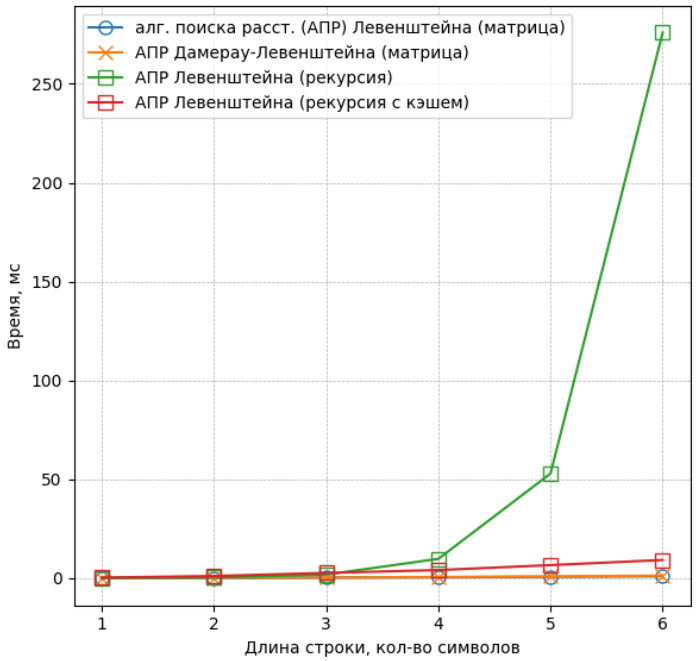
Таблица 4.1 – Результат замеров процессорного времени работы алгоритмов для строк длиной от 1 до 6 символов.

Length	**Л (матрица)**	**ДЛ (матрица)**	**Л (рекурсия)**	**Л (рекурсия с кэшем)**
1	0.100097	0.097736	0.095129	0.347810
2	0.176366	0.204006	0.399600	1.138200
3	0.340942	0.394079	1.860301	2.702778
4	0.512742	0.631439	9.813755	4.151481
5	0.741701	0.947312	52.949607	6.649858
6	1.012530	1.341707	275.854151	9.246883

На рисунке 4.1 представлены графики, построенные на основе табличных данных.



а) Логарифмическая шкала



б) Линейная шкала

Рисунок 4.1 – Зависимость процессорного времени работы алгоритмов от длины входных строк

4.3 Оценка потребляемой памяти

4.3.1 Рекурсивный алгоритм Левенштейна

Глубина рекурсии в рекурсивных алгоритмах нахождения расстояния Левенштейна в худшем случае будет составлять

$$(len(S_1) + len(S_2)). \tag{4.1}$$

Также на каждом уровне создаются новые строки, что в худшем случае требует $sizeof(S_1) + sizeof(S_2)$ байт на каждом уровне рекурсии. При этом каждый рекурсивный вызов требует хранения двух локальных переменных длин входных строк. С учётом того, что в данном алгоритме используются три глобальные переменные $INSERT\_COST$, $DELETE\_COST$, $REPLACE\_COST$, которые в сумме занимают $3 \cdot sizeof(int)$ байт, итоговый объем будет выражен формулой (4.2).

$$(len(S_1)+len(S_2)) \cdot (sizeof(S_1)+sizeof(S_2)+2 \cdot sizeof(int))+3 \cdot sizeof(int) \tag{4.2}$$

4.3.2 Рекурсивный алгоритм Левенштейна с кэшем

При каждом рекурсивном вызове для хранения локальных переменных $length1$, $length2$ и $distance$ требуется $3 \cdot sizeof(int)$ байт, для хранения кортежа из двух целочисленных значений необходимо $key\_size = sizeof(tuple) + 2 \cdot sizeof(int)$ байт, где $tuple$ - объект кортежа.

Общее количество уникальных рекурсивных вызовов равно

$$(len(S_1) + 1) \cdot (len(S_2) + 1), \tag{4.3}$$

а для хранения одной пары «ключ-значение», где ключ — кортеж из двух целых чисел, а значение — одно целое число, требуется количество байт, вычисляемое по формуле (4.4).

$$key\_value\_size = (key\_size + sizeof(int)). \tag{4.4}$$

Тогда в худшем случае для словаря и его содержимого потребуется

объем памяти, выраженный формулой (4.5), где *dict* — объект словаря.

$$memo\_size = sizeof(dict) + (len(S_1) + 1) \cdot (len(S_2) + 1) \cdot key\_value\_size. \quad (4.5)$$

С учётом того, что в данном алгоритме используются три глобальные переменные $INSERT\_COST$, $DELETE\_COST$, $REPLACE\_COST$, которые в сумме занимают $3 \cdot sizeof(int)$ байт, итоговый объем памяти, необходимый в худшем случае, будет выражен формулой (4.6).

$$((len(S_1) + len(S_2)) \cdot (sizeof(S_1) + sizeof(S_2) + 3 \cdot sizeof(int) + key\_size) + memo\_size + 3 \cdot sizeof(int)). \quad (4.6)$$

4.3.3 Матричный алгоритм Левенштейна

В данном алгоритме для хранения локальных переменных длин строк S_1, S_2 и параметра *cost* необходимо $3 \cdot sizeof(int)$ байт. Для хранения матрицы результатов M необходимо $(len(S_1) + 1) \cdot (len(S_2) + 1) \cdot sizeof(int)$ байт. Также в данном алгоритме используются глобальные целочисленные переменные $INSERT\_COST$, $DELETE\_COST$, $REPLACE\_COST$, которые в сумме занимают $3 \cdot sizeof(int)$ байт. Итоговый объем памяти выражен формулой (4.7).

$$(len(S_1) + 1) \cdot (len(S_2) + 1) \cdot sizeof(int) + 6 \cdot sizeof(int) \quad (4.7)$$

4.3.4 Матричный алгоритм Дамерау — Левенштейна

В данном алгоритме для хранения локальных переменных длин строк S_1, S_2 и параметра *cost* необходимо $3 \cdot sizeof(int)$ байт. Для хранения матрицы результатов M необходимо $(sizeof(S_1) + 1) \cdot (sizeof(S_2) + 1) \cdot sizeof(int)$ байт. Также в данном алгоритме используются глобальные целочисленные переменные $INSERT\_COST$, $DELETE\_COST$, $REPLACE\_COST$ и

$TRANSPOSITION\_COST$, которые в сумме занимают $4 \cdot sizeof(int)$ байт. Итоговый объем памяти выражен формулой (4.8).

$$(sizeof(S_1) + 1) \cdot (sizeof(S_2) + 1) \cdot sizeof(int) + 7 \cdot sizeof(int) \tag{4.8}$$

4.4 Вывод

На основе результатов замера процессорного времени работы, можно сделать вывод, что матричные алгоритмы нахождения расстояний Левенштейна и Дамерау — Левенштейна имеют схожее время работы (в сравнении с рекурсивными алгоритмами разница между матричными алгоритмами крайне мала) и потребляемое количество памяти, т. к. алгоритм нахождения расстояния Дамерау — Левенштейна использует всего на одну переменную больше ($TRANSPOSITION\_COST$).

Простой рекурсивный алгоритм нахождения расстояния Левенштейна имеет худшее время работы. Также можно подтвердить, что мемоизация действительно даёт улучшение производительности: для строк длиной 6 символов скорость работы рекурсивного алгоритма Левенштейна с кэшем в 30 раз больше скорости простого рекурсивного алгоритма Левентшейна. Но при использовании мемоизации увеличивается количество потребляемой памяти, т. к. необходимо хранить результаты предыдущих вызовов.

Стоит отметить, что рекурсивные реализации имеют аппаратное ограничение, так как ввод длинных строк может привести к переполнению стека из-за множественных рекурсивных вызовов.

Выбор между алгоритмами нахождения расстояний Левенштейна и Дамерау — Левенштейна зависит от необходимости учёта операции транспозиции соседних символов.

ЗАКЛЮЧЕНИЕ

Матричные алгоритмы Левенштейна и Дамерау —- Левенштейна имеют схожее время работы и потребление памяти, поскольку последний использует лишь одну дополнительную переменную для учёта транспозиций. Рекурсивный алгоритм Левенштейна обладает худшим временем работы, но использование мемоизации улучшает его производительность — для строк длиной 6 символов скорость работы увеличивается в 30 раз. Однако мемоизация требует дополнительных затрат памяти для хранения промежуточных результатов.

Рекурсивные реализации ограничены глубиной стека, что делает их непригодными для работы с длинными строками. Выбор между алгоритмами Левенштейна и Дамерау — Левенштейна зависит от необходимости учёта операции транспозиции.

Все задачи лабораторной работы выполнены:

- построены схемы рекурсивных и нерекурсивных алгоритмов поиска расстояния Левенштейна и Дамерау — Левенштейна;
- выполнена программная реализация перечисленных выше алгоритмов;
- проведён анализ процессорного времени работы и потребляемой памяти для реализаций алгоритмов;
- полученные результаты описаны и обоснованы.

Цель лабораторной работы достигнута: было проведено исследование различных реализаций алгоритмов нахождения расстояний Левенштейна и Дамерау — Левенштейна на основе разработанного ПО.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Ульянов А.* Ресурсно-эффективные компьютерные алгоритмы. Разработка и анализ. — Москва : Наука, 2010.
2. *Gusfield D.* Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. — Cambridge : Cambridge University Press, 1997.
3. MicroPython [Электронный ресурс]. — Режим доступа: `https://micropython.org/`(дата обращения: 01.11.2024).
4. time – time related functions [Электронный ресурс]. — Режим доступа: `https://docs.micropython.org/en/latest/library/time.html#module-time` (дата обращения: 01.11.2024).
5. NUCLEO-F767ZI – STM32 Nucleo-144 development board [Электронный ресурс]. — Режим доступа: `https://www.st.com/en/evaluation-tools/nucleo-f767zi.html#documentation` (дата обращения: 01.11.2024).