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 Расстояние Левенштейна и Дамерау – Левенштейна

Студент Чепиго Д.С.

Группа ИУ7-54Б

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

Москва — 2022 г.

Содержание

Введение

Расстояние Левенштейна (редакционное расстояние) – метрика, измеряющая разность между двумя последовательностями символов. Она определяется как минимальное количество односимвольных операций (а именно вставки, удаления, замены), необходимых для превращения одной последовательности символов в другую. Впервые задачу нахождения редакционного расстояния поставил в 1965 году советский математик Владимир Левенштейн при изучении последовательностей, состоящих из 0 и 1 [1].

Расстояние Дамерау – Левенштейна (названо в честь учёных Фредерика Дамерау и Владимира Левенштейна) является модификацией расстояния Левенштейна: к операциям вставки, удаления и замены символов, определённых в расстоянии Левенштейна добавлена операция транспозиции (перестановки) символов.

Расстояние Левенштейна и похожие расстояния активно применяются:

1) для исправления ошибок в слове (в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи);

2) для сравнения текстовых файлов утилитой `diff` и ей подобными (здесь роль «символов» играют строки, а роль «строк» — файлы);

3) в биоинформатике для сравнения последовательности белков.

Цель – изучить метод динамического программирования на примере редакционных расстояний.

Задачи лабораторной работы:

- изучение алгоритмов нахождения расстояния Левенштейна и Дамерау – Левенштейна;
- применение методов динамического программирования для реализации алгоритмов поиска расстояния Левештейна и Дамерау – Левенштейна;
- сравнительный анализ алгоритмов на основе экспериментальных данных;
- подготовка отчета по лабораторной работе.

1 Аналитическая часть

Расстояние Дамерау – Левенштейна между двумя строками – количество редакторских операций необходимых для преобразования одной строки в другую.

Редакторские операции:

- вставка (англ. insert);
- удаление (англ. delete);
- замена (англ. replace).

Для расстояния Дамерау – Левенштейна вводится редакторская операция транспозиция (англ. transposition).

В общем случае:

- $w(a,b) = 1$ – цена замены символа a на символ b, $a \neq b$;
- $w(\lambda,b) = 1$ – цена вставки символа b;
- $w(a,\lambda) = 1$ – цена удаления символа a;
- $w(ab, ba) == 1$ – цена транспозиции двух соседних символов;
- $w(a,a) = 0$ – цена совпадения.

1.1 Нерекурсивный алгоритм поиска расстояния Левенштейна

Пусть S_1 и S_2 – две строки длиной $\text{len}(S_1)$ и $\text{len}(S_2)$ соответственно, над некоторым конечным алфавитом.

Расстояние Левенштейна может быть найдено по формуле 1.1, которая задана как:

$$D(i,j) = \begin{cases} \max(i,j), & \text{если } \min(i,j) = 0, \\ \min\{ & \\ \quad D(i,j-1)+1, & \\ \quad D(i-1,j)+1, & \\ \quad D(i-1,j-1)+m(S_1[i],S_2[j]), & \text{иначе} \\ \} & \end{cases}, \quad (1.1)$$

где функция m определена как:

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

Для оптимизации нахождения расстояния используется матрица промежуточных значений. Ее размерность $(len(S_1)+1) \times (len(S_2)+1)$.

Значение в ячейке $matrix[i,j]$ равно значению $D(S_1[1...i], S_2[1...j])$. Первая строка и первый столбец тривиальны и совпадают с наибольшим значением индекса i или j ячейки.

Вся матрица (кроме первого столбца и первой строки) заполняется в со-

ответствии с формулой 1.3:

$$matrix[i][j] = min \begin{cases} matrix[i-1][j] + 1, \\ matrix[i][j-1] + 1, \\ matrix[i-1][j-1] + m(S_1[i], S_2[j]), \\ \begin{cases} matrix[i-2][j-2] + 1, & \text{если } i,j > 1; \\ & S_1[i] = S_2[j-1]; \\ & S_1[i-1] = S_2[j] \end{cases} \end{cases} . \tag{1.3}$$

В результате расстоянием Левенштейна будет ячейка матрицы с индексами $i = len(S_1)$ и $j = len(S_2)$.

1.2 Нерекурсивный алгоритм поиска расстояния Дамерау – Левенштейна

Расстояние Дамерау – Левенштейна может быть найдено по формуле 1.4, которая задана как:

$$D(i,j) = \begin{cases} \max(i,j), & \text{если } \min(i,j) = 0, \\ \min\{ & \\ \quad D(i,j-1) + 1, & \\ \quad D(i-1,j) + 1, & \\ \quad D(i-1,j-1) + m(S_1[i], S_2[j]), & \text{иначе} \\ \quad \begin{cases} D(i-2,j-2) + 1, & \text{если } i,j > 1; \\ & S_1[i] = S_2[j-1]; \\ & S_1[i-1] = S_2[j] \end{cases} & \\ \quad \infty, & \text{иначе} \\ \} & \end{cases} , \tag{1.4}$$

Формула выводится по тем же соображениям, что и формула 1.1, но с добавлением редакторской операции транспозиции.

В результате расстоянием Дамерау – Левенштейна будет ячейка матрицы с индексами $i = len(S_1)$ и $j = len(S_2)$.

1.3 Рекурсивный алгоритм поиска расстояния Дамерау – Левенштейна

Рекурсивный алгоритм реализует формулу 1.4. Функция D составлена из следующих соображений:

1) для перевода из пустой строки в пустую требуется ноль операций;

2) для перевода из пустой строки в строку S_1 требуется $|S_1|$ операций;

3) для перевода из строки S_1 в пустую требуется $|S_1|$ операций;

Для перевода из строки S_1 в строку S_2 требуется выполнить последовательно некоторое количество операций (удаление, вставка, замена, транспозиция) в некоторой последовательности.

Полагая, что S_1', S_2' и S_1'', S_2'' — строки S_1 и S_2 без последнего символа и без двух последних символов соответственно, цена преобразования из строки S_1 в строку S_2 может быть выражена как:

1) сумма цены преобразования строки S_1' в S_2 и цены проведения операции удаления, которая необходима для преобразования S_1' в S_1;

2) сумма цены преобразования строки S_1 в S_2' и цены проведения операции вставки, которая необходима для преобразования S_2' в S_2;

3) сумма цены преобразования из S_1' в S_2' и операции замены, предполагая, что S_1 и S_2 оканчиваются на разные символы;

4) цена преобразования из S_1' в S_2', предполагая, что S_1 и S_2 оканчиваются на один и тот же символ.

5) сумма цены преобразования S_1'' в S_2'', при условии, что два последних символа S_1 равны двум последним символам S_2 после транспозиции;

Минимальной ценой преобразования будет минимальное значение из приведенных вариантов.

1.4 Рекурсивный с кешированием алгоритм поиска расстояния Дамерау – Левенштейна

Рекурсивный алгоритм заполнения можно оптимизировать по времени выполнения с использованием матричного алгоритма. Суть данного метода заключается в заполнении матрицы при выполнении рекурсии. В случае, если рекурсивный алгоритм выполняет прогон для данных, которые еще не были обработаны, результат нахождения расстояния заносится в матрицу. В случае, если обработанные ранее данные встречаются снова, то есть ячейка матрицы уже заполнена, для них расстояние не находится и алгоритм переходит к следующему шагу.

2 Конструкторская часть

2.1 Разработка алгоритма нахождения расстояния Левенштейна

На рисунке 2.1 приведена схема итеративного алгоритма нахождения расстояния Левенштейна.

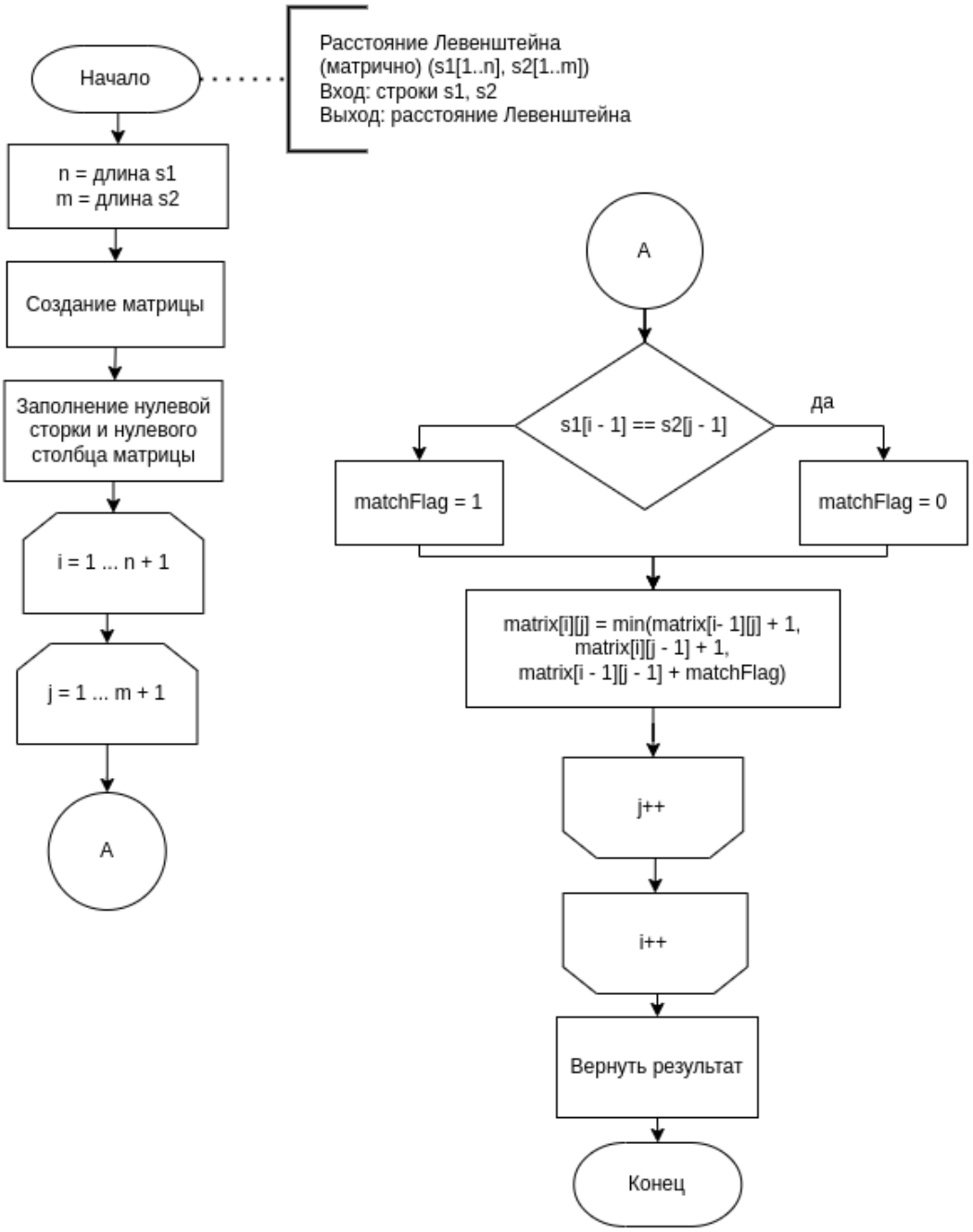

Рисунок 2.1 – Схема итеративного алгоритма нахождения расстояния Левенштейна

2.2 Разработка алгоритмов нахождения расстояния Дамерау – Левенштейна

На рисунке 2.2 приведена схема алгоритма нахождения расстояния Дамерау – Левенштейна с заполнением матрицы.

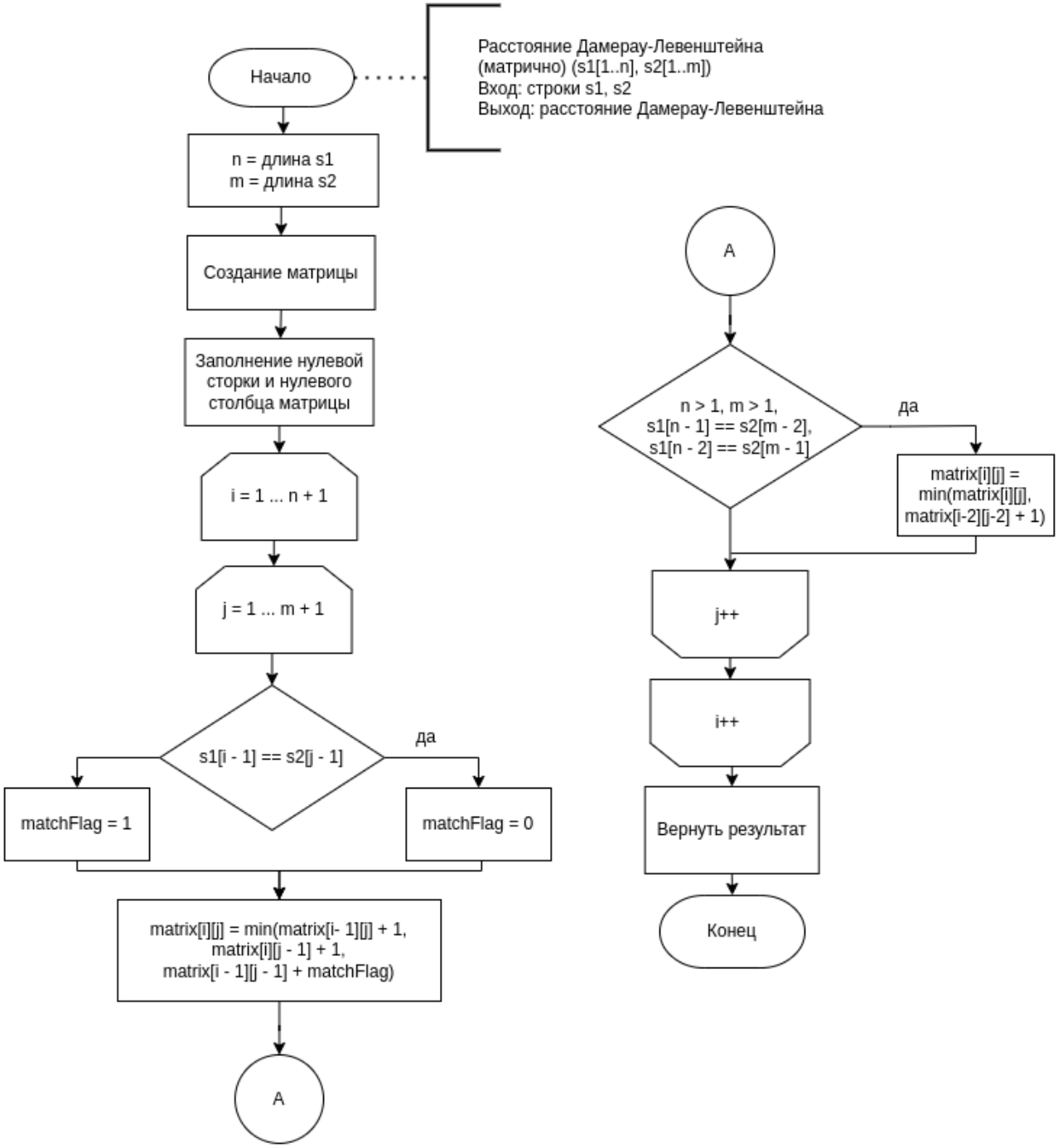

Рисунок 2.2 – Схема итеративного алгоритма нахождения расстояния Дамерау – Левенштейна

На рисунке 2.3 приведена схема рекурсивного алгоритма нахождения расстояния Дамерау – Левенштейна без кеширования.

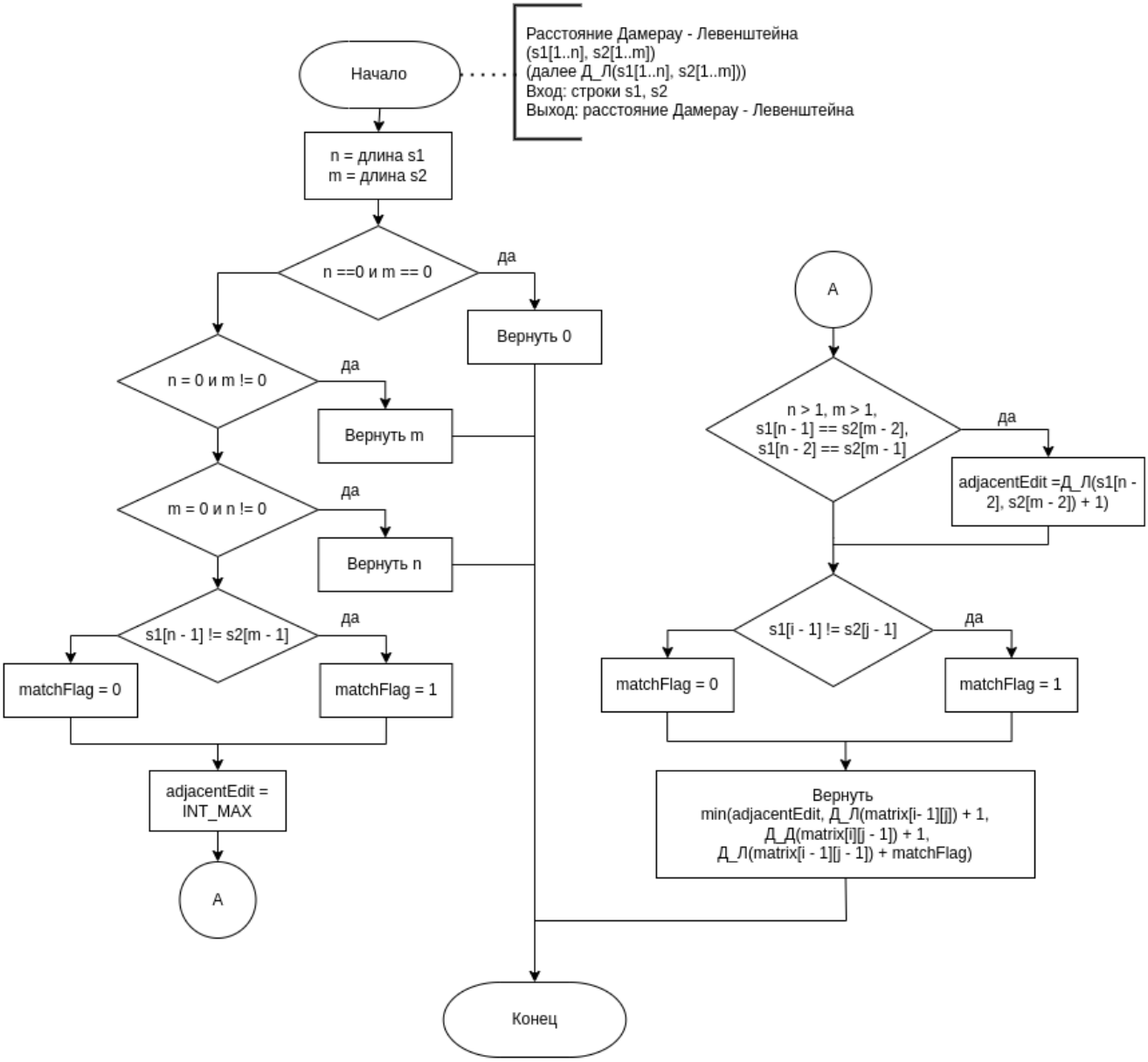

Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Дамерау – Левенштейна без кеширования

На рисунке 2.4 приведена схема рекурсивного алгоритма нахождения расстояния Дамерау – Левенштейна с кешированием.

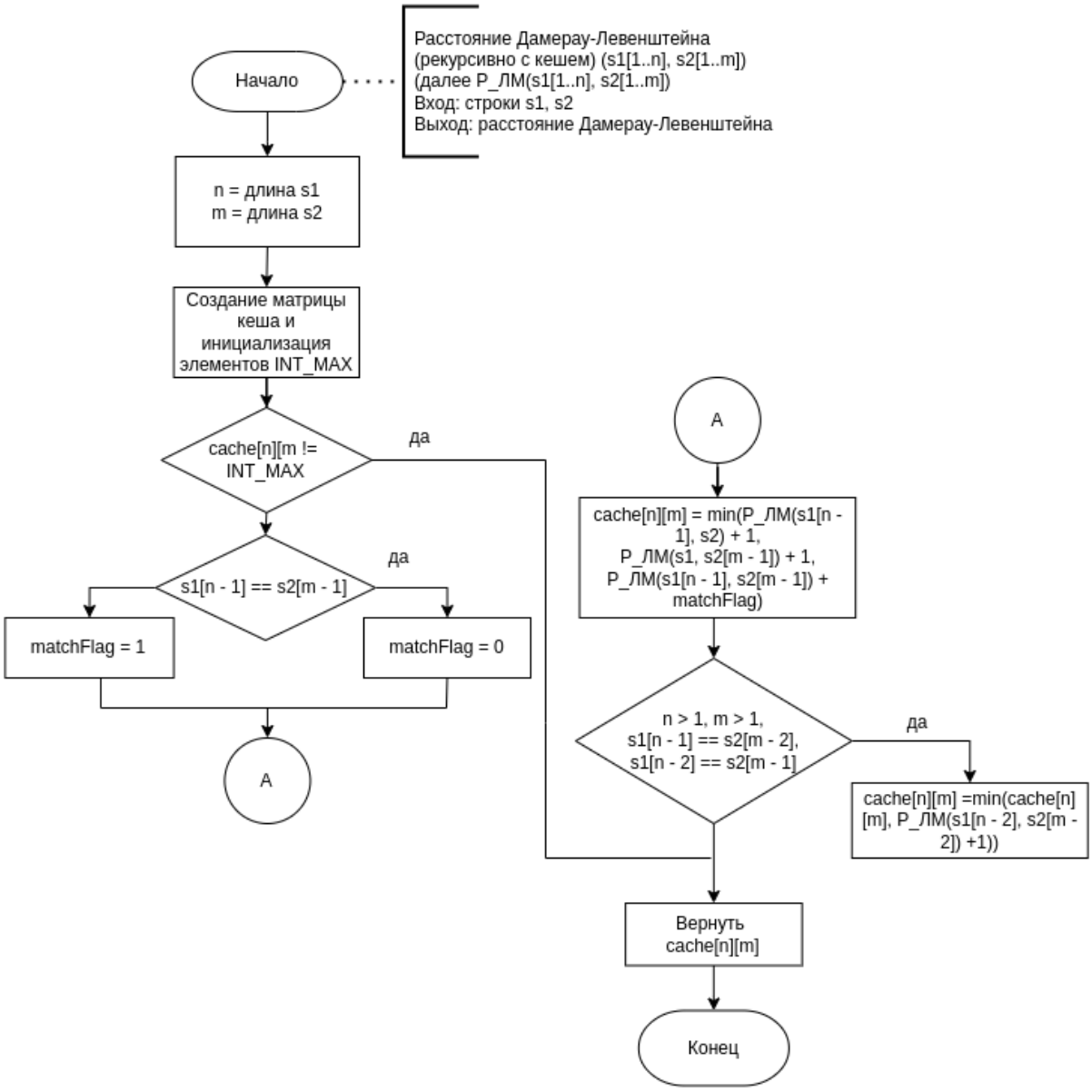

Рисунок 2.4 – Схема рекурсивного алгоритма нахождения расстояния Дамерау – Левенштейна с кешированием

3 Технологическая часть

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

3.1 Требования к ПО

К программе предъявляется ряд требований:

- на вход подаются две строки на русском или английском языке в любом регистре;
- на выходе – искомое расстояние для трех методов, матрица расстояний для итеративного метода и время выполнения в тиках.

3.2 Средства реализации

В качестве языка программирования для реализации лабораторной работы был выбран C++ – компилируемый, статически типизированный язык программирования общего назначения [2].

Данный выбор обусловлен поддержкой языком парадигмы объектно – ориентированного программирования и наличием методов для замера процессорного времени.

Время выполнения реализации алгоритмов было замерено с помощью ассемблерной вставки и измеряется в тиках.

3.3 Сведения о модулях программы

Программа состоит из трех программных модулей:

1) main.cpp – главный модуль программы, содержащий функцию main, с которой начинается выполнение программы;
2) algorithm.cpp, algorithm.h – модуль с реализацией алгоритмов;

3) timer.cpp, timer.h – модуль для замера времени.

3.4 Реализация алгоритмов

В листинге 3.1 приведена реализация итеративного алгоритма нахождения расстояния Дамерау – Левенштейна, а также вспомогательные функции.

Листинг 3.1 – Итеративный алгоритм

```
void distanceSolver::fillMatrixDamerauLevenshtein()
{
    for (size_t i = 0; i < row; i++)
        matrix[i][0] = i;

    for (size_t i = 0; i < column; i++)
        matrix[0][i] = i;

    for (size_t i = 1; i < row; i++)
    {
        for (size_t j = 1; j < column; j++)
        {
            if (firstString[i - 1] == secondString[j - 1])
            {
                matrix[i][j] = matrix[i-1][j-1];
            }
            else
            {
                matrix[i][j] = 1 + min(min(matrix[i-1][j], matrix[i][j-1]),
                    matrix[i-1][j-1]);

                if (checkAdjacentSymb(i, j))
                    matrix[i][j] = min(matrix[i][j], matrix[i - 2][j - 2] + 1);
            }
        }
    }

}

int distanceSolver::iterativeDamerauLevenshtein()
{
    allocateMatrix();
    fillMatrixDamerauLevenshtein();

    return matrix[row - 1][column - 1];
}
```

В листинге 3.2 приведена реализация рекурсивного алгоритма нахождения расстояния Дамерау – Левенштейна, а также вспомогательные функции.

Листинг 3.2 – Рекурсивный алгоритм без кеша

```
int distanceSolver::recursive(int firstLen, int secondLen)
{
    if (firstLen == 0 && secondLen == 0)
        return 0;

    if (firstLen == 0)
        return secondLen;

    if (secondLen == 0)
        return firstLen;

    int matchFlag = (firstString[firstLen - 1] == secondString[secondLen - 1])
       ? 0 : 1;
    int adjacentEdit = INT_MAX;

    if (checkAdjacentSymb(firstLen, secondLen))
        adjacentEdit = recursive(firstLen - 2, secondLen - 2) + 1;

    return min(min(recursive(firstLen - 1, secondLen) + 1,
                recursive(firstLen - 1, secondLen - 1) + matchFlag),
                min(recursive(firstLen, secondLen - 1) + 1, adjacentEdit));
}

int distanceSolver::recursiveDamerauLevenshtein()
{
    return recursive(firstString.length(), secondString.length());
}
```

В листинге 3.3 приведена реализация рекурсивного алгоритма нахождения расстояния Дамерау – Левенштейна с кешированием, а также вспомогательные функции.

Листинг 3.3 – Рекурсивный алгоритм с кешированием

```
int distanceSolver::recursiveCache(int firstLen, int secondLen)
{
    if (cache[firstLen][secondLen] != INT_MAX)
        return cache[firstLen][secondLen];

    int matchFlag = (firstString[firstLen - 1] == secondString[secondLen - 1])
        ? 0 : 1;

    cache[firstLen][secondLen] = min(min(recursiveCache(firstLen - 1,
        secondLen) + 1,
                recursiveCache(firstLen - 1, secondLen - 1) + matchFlag),
                recursiveCache(firstLen, secondLen - 1) + 1);

    if (checkAdjacentSymb(firstLen, secondLen))
            cache[firstLen][secondLen] = min(cache[firstLen][secondLen],
                                             recursiveCache(firstLen - 2,
                                                 secondLen - 2) + 1);


    return cache[firstLen][secondLen];
}

int distanceSolver::recursiveCacheDamerauLevenshtein()
{
    allocateCache();

    return recursiveCache(firstString.length(), secondString.length());
}
```

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- операционная система Ubuntu 22.04.1 LTS Linux x86_64 [3];
- память 8 ГБ;
- процессор Intel® Core™ i3-7130U.

Тестирование проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, окружением, а также непосредственно системой тестирования.

4.2 Пример работы программы

На рисунке 4.1 представлен пример работы программы. Вводятся две строки, в данном случае «dasha» и «arisah», размера 5 и 6 соответственно. Для даных строк расстояние Левенштейна равно 5, а расстояние Дамерау – Левенштейна 4. Это связано с редакторской операцией транспозиции для букв «a» и «h». Для итеративных алгоритмов также выводится матрица. Также выводится время выполнения каждого алгоритма в тиках.

```
dashori@fossa ~/P/A/l/s/code (develop)> ./a.out
Input first string: dasha
Input secong string: arisah

Iterative Levenshtein
Matrix:
0 1 2 3 4 5 6
1 1 2 3 4 5 6
2 1 2 3 4 4 5
3 2 2 3 3 4 5
4 3 3 3 4 4 4
5 4 4 4 4 4 5
Result: 5
Time: 69060 (ticks)

Iterative Damerau-Levenshtein
Matrix:
0 1 2 3 4 5 6
1 1 2 3 4 5 6
2 1 2 3 4 4 5
3 2 2 3 3 4 5
4 3 3 3 4 4 4
5 4 4 4 4 4 4
Result: 4
Time: 48763 (ticks)

Recursive Damerau-Levenshtein without cache
Result: 4
Time: 1674321 (ticks)

Recursive Damerau-Levenshtein with cache
Result: 4
Time: 82773 (ticks)
```

Рисунок 4.1 – Пример работы программы

4.3 Время выполнения реализации алгоритмов

Время работы реализации алгоритмов было замерено с помощи ассемблерной вставки 4.1. Полученное время измеряется в тиках.

Листинг 4.1 – Ассемблерная вставка

```
#include "time.h"

uint64_t tick(void)
{
    uint32_t high, low;
    __asm__ __volatile__ (
        "rdtsc\n"
        "movl %%edx, %0\n"
        "movl %%eax, %1\n"
        : "=r" (high), "=r" (low)
        :: "%rax", "%rbx", "%rcx", "%rdx"
        );

    uint64_t ticks = ((uint64_t)high << 32) | low;

    return ticks;
}
```

Замеры времени работы реализованных алгоритмов для определенной длины строк проводились 50 раз, при этом каждый раз строки генерировались случайно. В качестве результата, представленного в таблице 4.1, взято среднее время на каждой длине слова. Замеры времени для реализации рекурсивного алгоритма поиска расстояния Дамерау – Левенштейна проводились до длины строк 9. Эти данные показали характер роста функции трудоёмкости и значений времени – функция имеет больший характер роста, чем для остальных трех реализаций алгоритмов.

Таблица 4.1 – Результаты замеров времени в тиках

Длина строк	Итеративный (Левенштейн)	Итеративный	Рекурсивный с кешем	Рекурсивный
2	5778	5507	7488	5155
3	8599	9853	13208	24538
4	13064	20477	22773	118544
5	14039	16225	25821	385112
6	19453	22825	32964	1623606
7	9429	16164	13886	3112536
8	6611	7752	11061	10516211
9	7515	10415	17934	57280850
10	8042	10699	14482	–
11	8325	10199	16205	–
12	9445	12121	19124	–
13	10948	14523	21885	–
14	10580	17325	26895	–
15	13822	19294	28440	–
16	15863	21484	32942	–
17	16613	23131	38527	–
18	19313	26357	40869	–
19	23353	29563	45762	–
20	26220	35045	55871	–
30	49478	78967	111383	–
40	113762	152507	246839	–
50	138558	193787	305259	–
60	193317	270063	440642	–
70	267411	380159	668288	–
80	362592	499571	796852	–
90	566693	895042	1210304	–
100	585323	752124	1281518	–

На рисунке 4.2 представлена зависимость времени работы от длины строк всех четырех реализаций алгоритмов. На рисунке 4.3 представлена зависимость времени работы от длины строк тех же реализаций алгоритмов, кроме реализации рекурсивного алгоритма поиска расстояния Дамерау – Левенштейна.

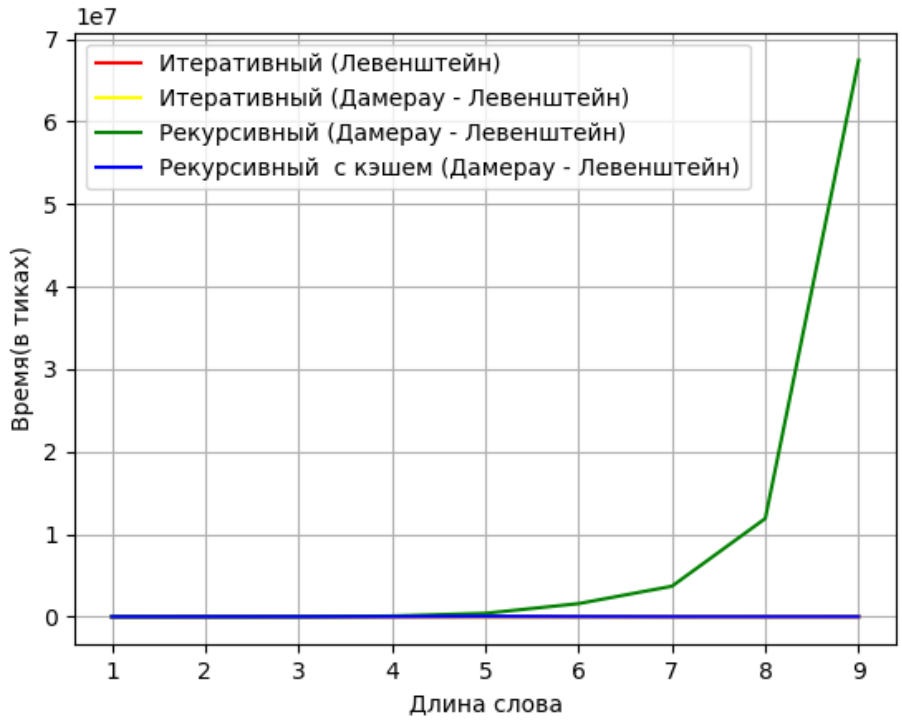

Рисунок 4.2 – Зависимость времени работы от длины строк алгоритмов поиска расстояния Левенштейна и Дамерау – Левенштейна

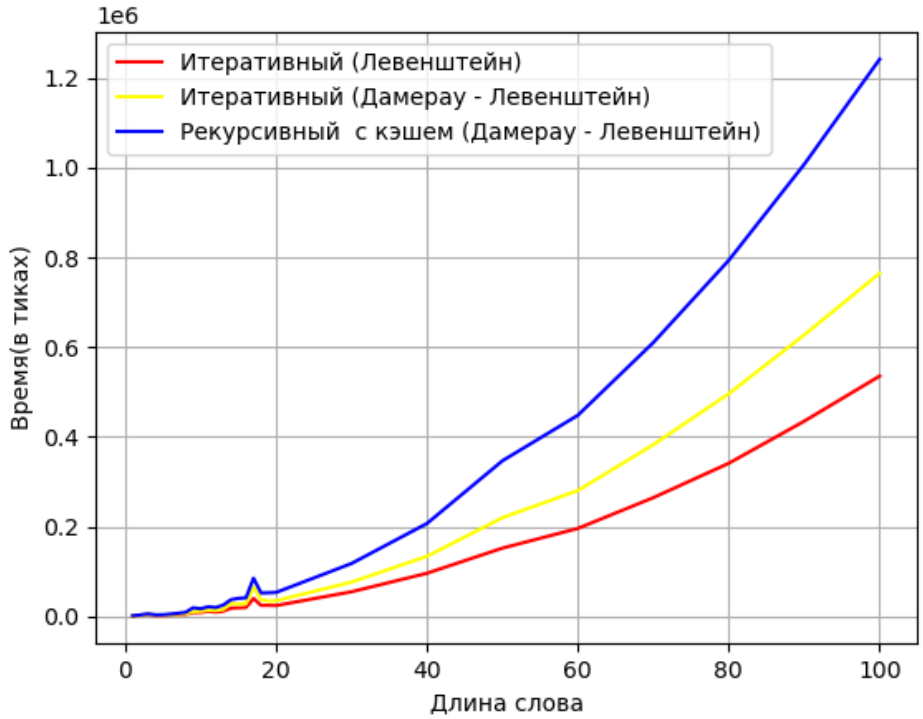

Рисунок 4.3 – Зависимость времени работы от длины строк рекурсивного алгоритма поиска расстояния Дамерау – Левенштейна и итеративных алгоритмов поиска расстояния Левенштейна и Дамерау – Левенштейна

4.4 Использование памяти

Рассмотрим разницу рекурсивной и матричной реализаций алгоритмов нахождения расстояния Дамерау – Левенштейна.

Пусть длина строки S_1 – n, длина строки S_2 - m, тогда затраты памяти на приведенные выше алгоритмы будут следующими:

- матричная реализация алгоритма поиска расстояния Левенштейна:
  - строки S_1, S_2 – $(m+n) \cdot sizeof(char)$;
  - длины строк – $2 \cdot sizeof(int)$;
  - матрица – $((m+1) \cdot (n+1)) \cdot sizeof(int)$;
  - текущая строка матрицы – $(n+1) \cdot sizeof(int)$;
  - вспомогательные переменные – $3 \cdot sizeof(int)$.

Итог: $(m+n) \cdot sizeof(char) + [(m+1) \cdot (n+1) + n + 6] \cdot sizeof(int)$;

- матричная реализация алгоритма поиска расстояния Дамерау – Левенштейна:
  - строки S_1, S_2 – $(m+n) \cdot sizeof(char)$;
  - длины строк – $2 \cdot sizeof(int)$;
  - матрица – $((m+1) \cdot (n+1)) \cdot sizeof(int)$;
  - текущая строка матрицы – $(n+1) \cdot sizeof(int)$;
  - вспомогательные переменные – $4 \cdot sizeof(int)$.

Итог: $(m+n) \cdot sizeof(char) + [(m+1) \cdot (n+1) + n + 7] \cdot sizeof(int)$;

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк.

- реализация рекурсивного алгоритма поиска расстояния Дамерау – Левенштейна с использованием кеша (для каждого вызова):
  - строки S_1, S_2 – $(m+n) \cdot sizeof(char)$;
  - длины строк – $2 \cdot sizeof(int)$;

– вспомогательные переменные – $2 \cdot sizeof(int)$;

– ссылка на матрицу – $sizeof(int*)$;

– адрес возврата.

Итог для каждого вывода: $(m+n) \cdot sizeof(char) + 5 \cdot sizeof(int) + 1 \cdot sizeof(int*)$;

Для всех вызовов память для хранения самой матрицы – $((m+1) \cdot (n+1)) \cdot sizeof(int)$.

Итог: $(m+n) \cdot [(m+n) \cdot sizeof(char) + 5 \cdot sizeof(int) + 1 \cdot sizeof(int*)] + (m+1) \cdot (n+1) \cdot sizeof(int)$;

- реализация рекурсивного алгоритма поиска расстояния Дамерау – Левенштейна(для каждого вызова):

  – строки S_1, S_2 – $(m+n) \cdot sizeof(char)$;

  – длины строк – $2 \cdot sizeof(int)$;

  – вспомогательная переменная – sizeof(int);

  – адрес возврата.

  Итог для каждого вывода: $(m+n) \cdot sizeof(char) + (6) \cdot sizeof(int) + 1 \cdot sizeof(int*)$;

  Итог: $(n+m) \cdot [(m+n) \cdot sizeof(char) + 6 \cdot sizeof(int) + 1 \cdot sizeof(int*)]$;

Результат замерных экспериментов реализованных алгоритмов показал, что самым быстрым алгоритмом является алгоритм поиска расстояния Левенштейна. Начиная с длины слова 10 он превосходит итеративный и рекурсивный с кешом алгоритм поиска расстояния Дамерау – Левенштейна в 1.5 и 2 раза соответственно. Рекурсивный алгоритм с заполнением матрицы превосходит простой рекурсивный.

При сравнении используемой памяти реализаций итеративные алгоритмов затрачивают меньше памяти, чем рекурсивный без кеширования начиная с $m+n = 6$. В свою очередь реализация итеративного алгоритма нахождения расстояния Левенштейна занимает меньше памяти, чем реализация итеративного алгоритма нахождения расстояния Дамерау – Левенштейна. Это связано с добавлением редакторской операции транспозиции. Самым затратным

по памяти является реализация рекурсивного алгоритма поиска расстояния Дамерау – Левенштейна с кешированием.

Заключение

Цель работы достигнута: изучен метод динамического программирования на примере редакционных расстояний.

В ходе выполнения лабораторной работы были решены все задачи:

- изучены алгоритмы нахождения расстояния Левенштейна и Дамерау – Левенштейна;
- применены методы динамического программирования для реализации алгоритмов;
- на основе полученных в ходе экспериментов данных были сделаны выводы по поводу эффективности всех реализованных алгоритмов;
- был подготовлен отчет по лабораторной работе.

Эксперименты показали, что наиболее затратный по времени рекурсивный алгоритм поиска расстояния Дамерау – Левенштейна без кеша, а наименее затратны итеративные алгоритмы. Менее затратными по памяти являются реализации итеративных алгоритмов. Самый затратный по памяти является оеализация рекурсивного алгоритма поиска расстояния Дамерау – Левенштейна с кешированием.

Литература

[1] Левенштейн В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов. – М.: Доклады АН СССР, 1965. Т. 163. С. 845–848.

[2] Язык программирования C++[Электронный ресурс]. Режим доступа: `https://isocpp.org/`(дата обращения: 20.09.2022)

[3] Операционная система Ubuntu 22.04 LTS[Электронный ресурс]. Режим доступа: `https://releases.ubuntu.com/jammy/`(дата обращения: 20.09.2022)