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 Редакционные расстояния

Студент Обревская В.В.

Группа ИУ7-52Б

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

Москва — 2022 г.

Оглавление

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

Расстояние Левенштейна и его обобщения активно применяются:

— для исправления ошибок в слове (в поисковых системах, базах данных, при вводе текста);

— в биоинформатике для сравнения генов, хромосом и белков.

Цель данной лабораторной работы – получение навыков динамического программирования на примере задачи поиска редакционных расстояний.

Задачи данной лабораторной работы:

1) изучение алгоритмов нахождения расстояния Левенштейна и Дамерау – Левенштейна;

2) реализация алгоритмов Левенштейна и Дамерау — Левенштейна;

3) выполнение замеров затрат реализаций алгоритмов по памяти;

4) выполнение замеров затрат реализаций алгоритмов по процессорному времени;

5) проведение сравнительного анализа двух нерекурсивных алгоритмов;

6) проведение сравнительного анализа алгоритмов поиска расстояния Дамерау – Левенштейна

1. Аналитическая часть

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

Расстояние Левенштейна между двумя строками – это минимальное количество операций, необходимых для превращения одной строки в другую.

Цены операций могут зависеть от вида операции (вставка, удаление, замена) и/или от участвующих в ней символов, отражая разную вероятность разных ошибок при вводе текста, и т. п. В общем случае:

— $w(a, b)$ — цена замены символа a на символ b.

— $w(\lambda, b)$ — цена вставки символа b.

— $w(a, \lambda)$ — цена удаления символа a.

Для решения задачи о редакционном расстоянии необходимо найти последовательность замен, минимизирующую суммарную цену. Расстояние Левенштейна является частным случаем этой задачи при

— $w(a, a) = 0$.

— $w(a, b) = 1,\ a \neq b$.

— $w(\lambda, b) = 1$.

— $w(a, \lambda) = 1$.

Для расстояния Дамерау – Левенштейна вводится редакторская операция – транспозиция. Цена транспозиции двух соседних символов – $w(ab, ba) = 1$.

1.1. Нерекурсивный алгоритм поиска расстояния Левенштейна

Расстояние Левенштейна между двумя строками S_1 и S_2 может быть вычислено по реккурентной формуле 1.1, где $|S_1|$ означает длину строки S_1, $S_1[i]$ — i-ый символ строки S_1, функция $D_{S_1,S_2}(i, j)$ определена как:

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

а функция m определена по формуле 1.2 как:

$$m(a,b) = \begin{cases} 0, & \text{если } a=b \\ 1, & \text{иначе} \end{cases}. \quad (1.2)$$

Для оптимизации нахождения расстояния Левенштейна используют матрицу в целях хранения соответствующих промежуточных значений. В таком случае алгоритм представляет собой построчное заполнение матрицы $M_{(|S_1|+1),(|S_2|+1)}$ значениями $D(i,j)$. В результате расстоянием Левенштейна будет ячейка матрицы с индексами $i=|S_1|$ и $j=|S_2|$.

1.2. Нерекурсивный алгоритм поиска расстояния Дамерау – Левенштейна

Является модификацией расстояния Левенштейна – добавлена операции транспозиции, то есть перестановки, двух символов.

Расстояние Дамерау — Левенштейна может быть найдено по формуле 1.3, которая задана как

$$d_{a,b}(i,j) = \begin{cases} \max(i,j), & \text{если } \min(i,j) = 0, \\ \min\{ & \\ \quad d_{a,b}(i,j-1) + 1, & \\ \quad d_{a,b}(i-1,j) + 1, & \\ \quad d_{a,b}(i-1,j-1) + m(a[i],b[j]), & \text{иначе} \\ \quad \begin{cases} d_{a,b}(i-2,j-2) + 1, & \text{если } i,j > 1; \\ & a[i] = b[j-1]; \\ & b[j] = a[i-1] \\ \infty, & \text{иначе} \end{cases} & \\ \} & \end{cases}, \tag{1.3}$$

Формула выводится по тем же соображениям, что и формула (1.1).

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

1.3. Рекурсивный алгоритм поиска расстояния Дамерау – Левенштейна

Рекурсивный алгоритм реализует формулу 1.3. Функция D составлена из следующих соображений:

— для перевода пустой строки в пустую требуется 0 операций;

— для перевода пустой строки в непустую строку S_1 требуется $|S_1|$ операций;

— для перевода непустой 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).

Полагая, что S_1', S_2' и S_1'', S_2'' строки S_1 и S_2 без одного и 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 вставки, которая необходима для преобразования S_2' в S_2;

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

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 из 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 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. Схема алгоритма нахождения расстояния Левенштейна

На рисунке 2.1 приведена схема нерекурсивного алгоритма поиска расстояния Левенштейна.

2.2. Схемы алгоритмов нахождения расстояния Дамерау – Левенштейна

На рисунке 2.2 приведена схема нерекурсивного алгоритма поиска расстояния Дамерау – Левенштейна.

Схема рекурсивного алгоритма поиска расстояния Дамерау – Левенштейна представлена на рисунке 2.3.

Схема рекурсивного с кешированием алгоритма поиска расстояния Дамерау – Левенштейна представлена на рисунке 2.4.

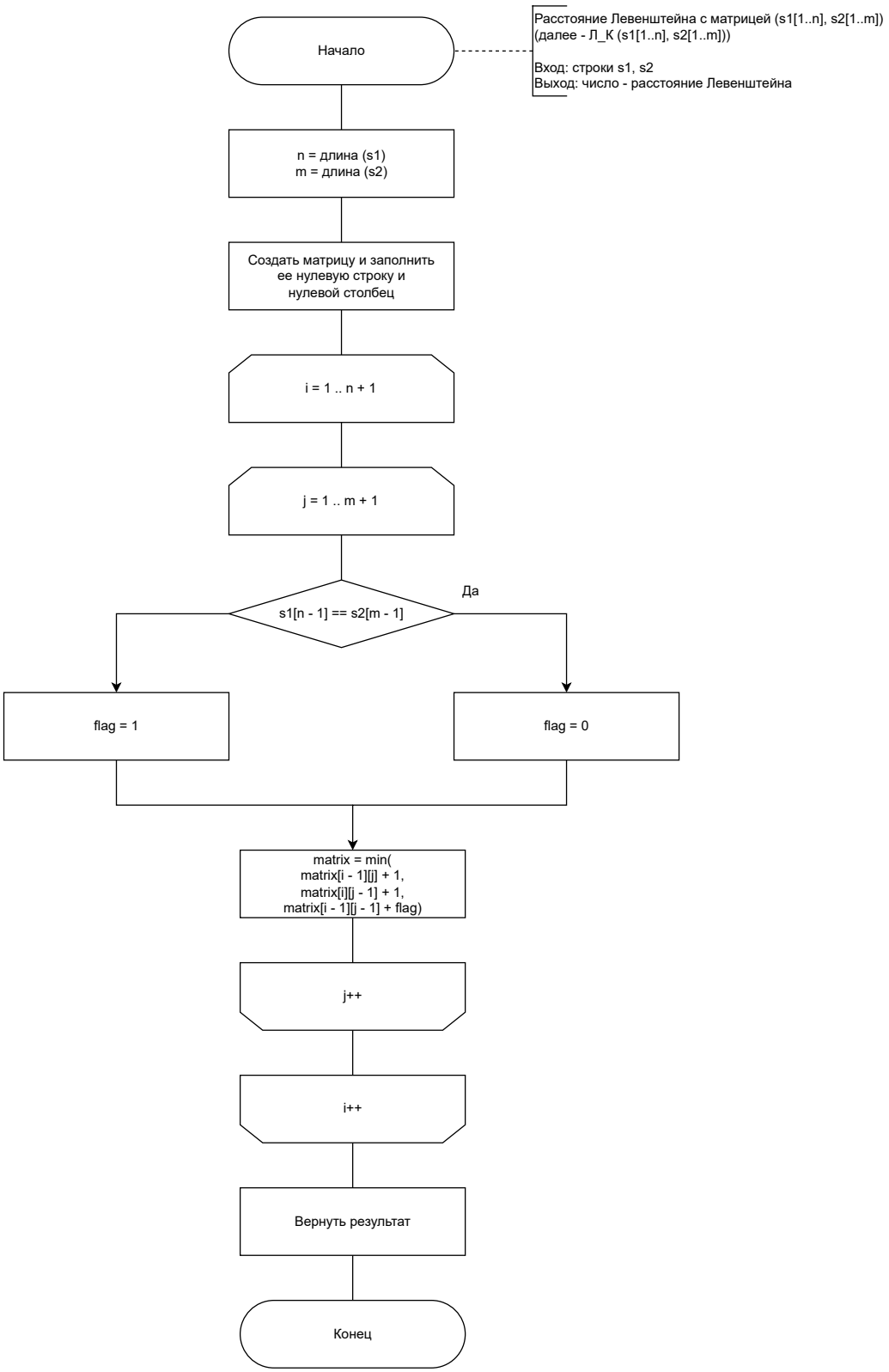

Рисунок 2.1 – Схема нерекурсивного алгоритма поиска расстояния Левенштейна

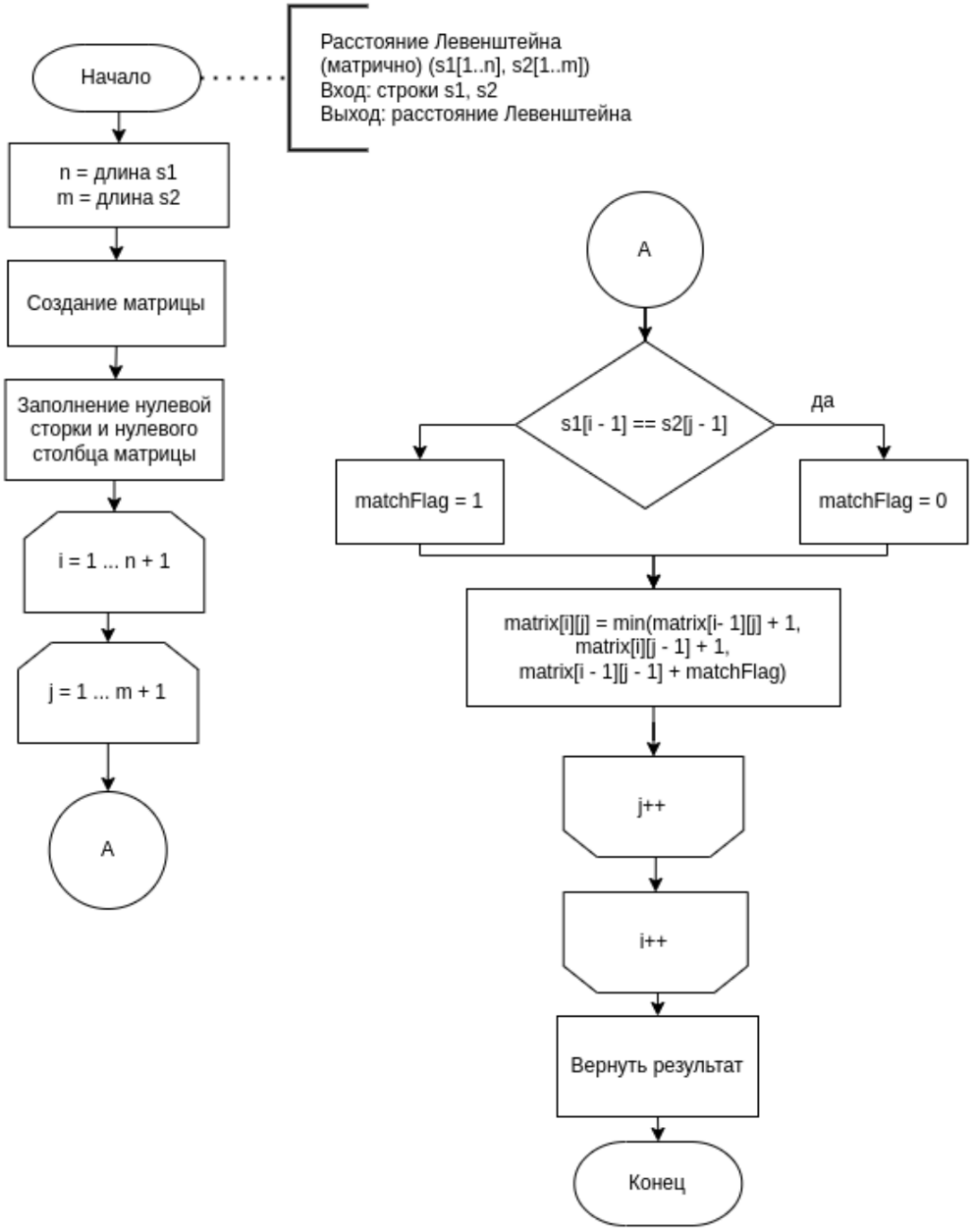

Рисунок 2.2 – Схема нерекурсивного алгоритма поиска расстояния Дамерау – Левенштейна

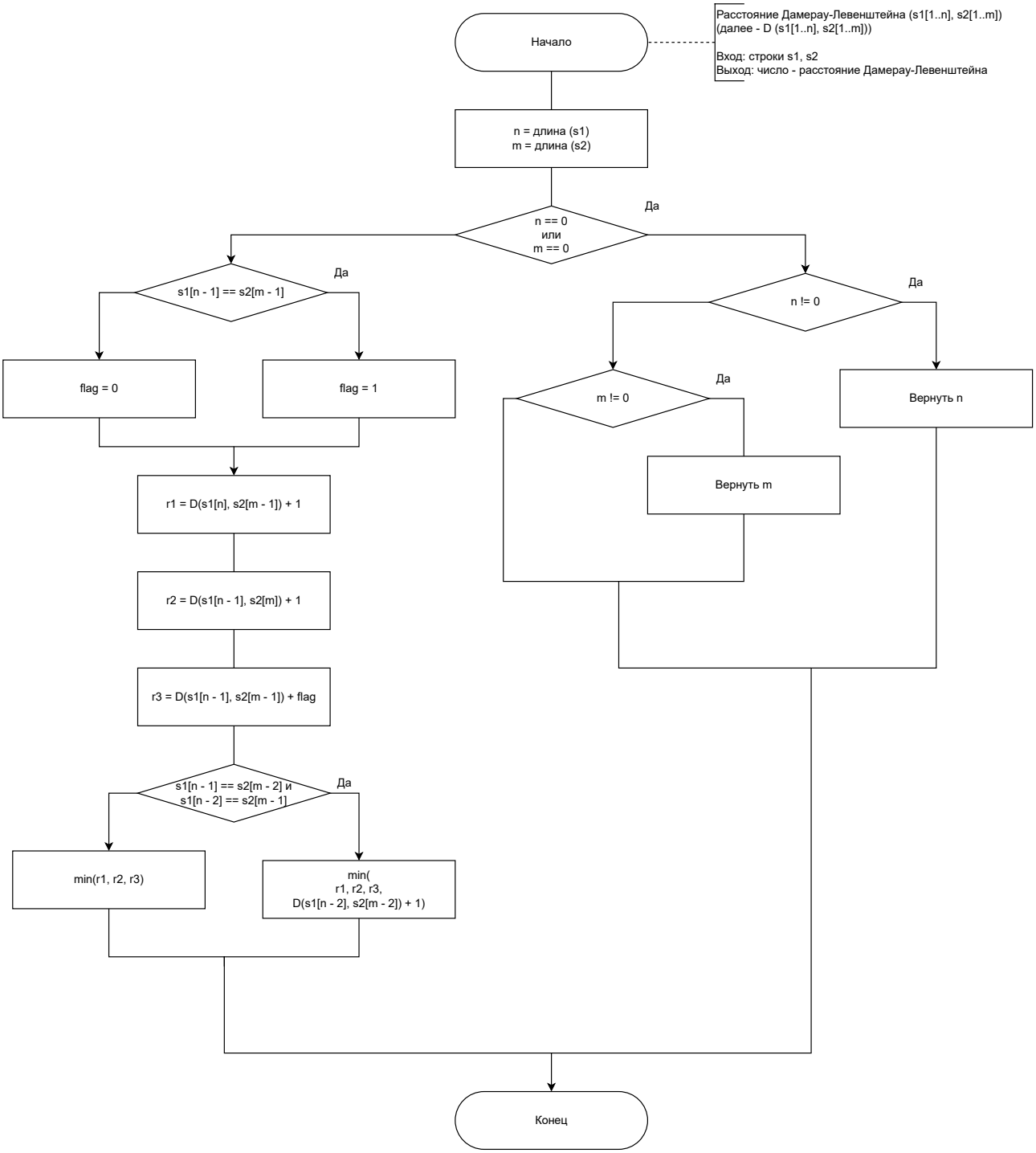

Рисунок 2.3 – Схема рекурсивного алгоритма поиска расстояния Дамерау – Левенштейна

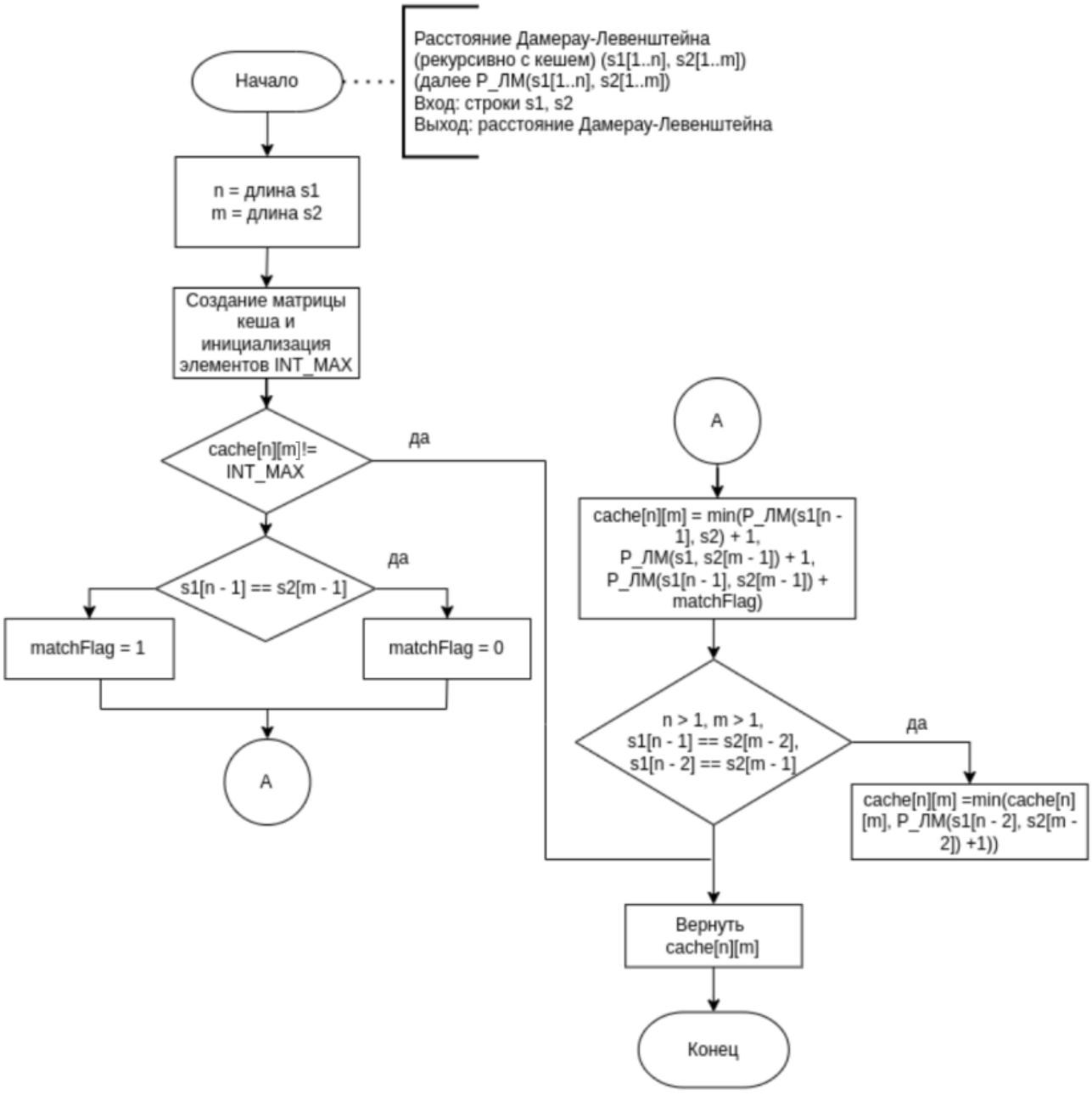

Рисунок 2.4 – Схема рекурсивного с кешированием алгоритма поиска расстояния Дамерау – Левенштейна

3. Технологическая часть

3.1. Требования к ПО

К программе предъявляется ряд требований.

— на вход подаётся две строки на английском или русском;

— на выходе – искомое расстояние для всех четырёх методов и время их 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.

3.3. Реализация алгоритмов

В листингах 3..1 – 3..7 приведены реализации алгоритмов нахождения расстояний Левенштейна и Дамерау – Левенштейна.

Листинг 3..1 – Реализация нерекурсивного алгоритма поиска расстояния Левенштейна (начало)

```
int levenshtein_distance(const wstring *s1, const wstring *s2)
{
    int n = (*s1).length();
    int m = (*s2).length();
    if (n == 0)
        return m;
    if (m == 0)
        return n;
```

Листинг 3..2 – Реализация нерекурсивного алгоритма поиска расстояния Левенштейна (конец)

```
9       int ans;
10      n++;
11      m++;
12
13      matrix_t matrix;
14      create_matrix(&matrix, n, m);
15
16      for (int i = 0; i < n; i++) {
17          for (int j = 0; j < m; j++) {
18              if (i * j == 0)
19                  matrix[i][j] = i + j;
20          }
21      }
22      bool turn = 0;
23      for (int i = 1; i < n; i++) {
24          for(int j = 1; j < m; j++) {
25              turn = (*s1)[i - 1] == (*s2)[j - 1] ? 0 : 1;
26              matrix[i][j] = min(min(matrix[i - 1][j] + 1, matrix[i][j
                    - 1] + 1), matrix[i - 1][j - 1] + turn);
27          }
28      }
29
30      ans = matrix[n - 1][m - 1];
31      delete_matrix(&matrix, n);
32      return ans;
```

Листинг 3..3 – Реализация нерекурсивного алгоритма поиска расстояния Дамерау – Левенштейна (начало)

```
1  int damerau_levenshtein_matr_distance(const wstring *s1, const
       wstring *s2)
2  {
3       int n = (*s1).length() + 1;
4       int m = (*s2).length() + 1;
5       if (n == 0)
6           return m;
7       if (m == 0)
8           return n;
```

Листинг 3..4 – Реализация нерекурсивного алгоритма поиска расстояния Дамерау – Левенштейна (конец)

```
    int ans;
    n++;
    m++;

    matrix_t matrix;
    create_matrix(&matrix, n, m);

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (i * j == 0)
                matrix[i][j] = i + j;
        }
    }
    bool turn = 0;
    for (int i = 1; i < n; i++) {
        for(int j = 1; j < m; j++) {
            turn = (*s1)[i - 1] == (*s2)[j - 1] ? 0 : 1;
            if ((i > 1 && j > 1) && ((*s1)[i-1] == (*s2)[j-2] && (*
               s1)[i-2] == (*s2)[j-1]) && (turn == 1))
                matrix[i][j] = min(min(matrix[i - 1][j] + 1, matrix[
                   i][j - 1] + 1), min(matrix[i - 1][j - 1] + turn,
                   matrix[i - 2][j - 2] + 1));
            else
                matrix[i][j] = min(min(matrix[i - 1][j] + 1, matrix[
                   i][j - 1] + 1), matrix[i - 1][j - 1] + turn);
        }
    }

    ans = matrix[n - 1][m - 1];
    delete_matrix(&matrix, n);
    return ans;
}
```

Листинг 3..5 – Реализация рекурсивного алгоритма поиска расстояния Дамерау – Левенштейна

```
int damerau_levenshtein_rec_distance(const std::wstring *s1, int n,
    const std::wstring *s2, int m)
{
    if (n == 0)
        return m;
    if (m == 0)
        return n;

    int ans = damerau_levenshtein_rec_distance(s1, n - 1, s2, m) +
       1;
    ans = min(ans, damerau_levenshtein_rec_distance(s1, n,
                                                    s2,
                                                    m - 1) + 1);

    bool t = (*s1)[n - 1] == (*s2)[m - 1] ? 0 : 1;
    int dist = damerau_levenshtein_rec_distance(s1, n - 1,
                                                s2, m - 1) + t;

    ans = min(ans, dist);
    if (n > 1 && m > 1)
        if ((*s1)[n - 2] == (*s2)[m - 1] && (*s1)[n - 1] == (*s2)[m
           - 2])
            dist = damerau_levenshtein_rec_distance(s1, n - 2,
                                                    s2, m - 2) + 1;
    ans = min(ans, dist);
    return ans;
}
```

Листинг 3..6 – Реализация рекурсивного с кешированием алгоритма поиска расстояния Дамерау – Левенштейна (начало)

```
1
2  int rec_cache(matrix_t *matrix, const wstring *s1, size_t i, const
       wstring *s2, size_t j)
3  {
4      int dist1, dist2, dist3;
5      int ans;
6
7      if ((dist1 = (*matrix)[i][j - 1]) == -1)
8          dist1 = rec_cache(matrix, s1, i, s2, j - 1);
9      if ((dist2 = (*matrix)[i - 1][j]) == -1)
10         dist2 = rec_cache(matrix, s1, i - 1, s2, j);
11     if ((dist3 = (*matrix)[i - 1][j - 1]) == -1)
12         dist3 = rec_cache(matrix, s1, i - 1, s2, j - 1);
13     if((*s1)[i - 1] != (*s2)[j - 1])
14         dist3++;
15     dist1++;
16     dist2++;
17     ans = min(min(dist1, dist2), dist3);
18
19     if (i > 1 && j > 1){
20         if ((*s1)[i - 1] == (*s2)[j - 2] && (*s1)[i - 2] == (*s2)[j
               - 1])
21             if ((dist1 = (*matrix)[i - 2][j - 2]) == -1)
22                 dist1 = rec_cache(matrix, s1, i - 2, s2, j - 2);
23         dist1++;
24     }
25     ans = min(ans, dist1);
26
27     (*matrix)[i][j] = ans;
28     return ans;
29 }
30
31 int damerau_levenshtein_rec_cache_distance(const std::wstring *s1,
       int n, const std::wstring *s2, int m)
32 {
33     if (n == 0)
34         return m;
35     if (m == 0)
36         return n;
```

Листинг 3..7 – Реализация рекурсивного с кешированием алгоритма поиска расстояния Дамерау – Левенштейна (конец)

```
37      matrix_t matrix;
38      create_matrix(&matrix, n + 1, m + 1);
39
40      for (int i = 0; i <= n; i++) {
41          for (int j = 0; j <= m; j++) {
42              if (i * j == 0)
43                  matrix[i][j] = i + j;
44              else
45                  matrix[i][j] = -1;
46          }
47      }
48
49      int ans = rec_cache(&matrix, s1, n, s2, m);
50      delete_matrix(&matrix, n + 1);
51      return ans;
52  }
```

3.4. Тестирование

В таблице 3.1 приведены тесты для алгоритмов вычисления расстояний Левенштейна и Дамерау – Левенштейна. Все тесты пройдены успешно.

Таблица 3.1 – Тесты

№	Строка 1	Строка 2	Ожидаемый результат	
			Левентшейн	Дамерау – Левентшейн
1	∅	∅	0	0
2	dog	∅	3	3
3	∅	bread	5	5
4	flower	flower	0	0
5	space	spice	1	1
6	collect	bargain	7	7
7	act	action	3	3
8	abcd	ybed	2	2
9	information	education	6	6
10	text	tetx	2	1

4. Экспериментальная часть

4.1. Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование, следующие.

— Операционная система Fedora linux 36 [3].

— Оперативная память: 16 Гб.

— Процессор: AMD Ryzen 7 5800U with Radeon Graphics, 1901 МГц, ядер: 8, логических процессоров: 16.

4.2. Демонстрация работы программы

На рисунке 4.1 представлен результат работы программы, если на вход поступили строчки *wood* и *blood*.

```
Input string 1:
wood
Input string 2:
blood

Levenshtein distance:
Answer: 2
Time: 19 ticks

Damerau-Levenshtein distance:
Answer: 2
Time: 6 ticks

Damerau-Levenshtein recursive distance:
Answer: 2
Time: 26 ticks

Damerau-Levenshtein recursive with cache distance:
Answer: 2
Time: 6 ticks
```

Рисунок 4.1 – Пример работы программы

4.3. Измерение процессорного времени выполнения реализаций алгоритмов

Время работы реализации алгоритмов было замерено с помощью функции clock. Полученное время измеряется в тиках, было замерено на компьютере подключенному к сети.

Результаты замеров в тиках приведены в таблице 4.1. В данной таблице для значений, для которых тестирование не выполнялось, в поле результата находится NaN.

Таблица 4.1 – Таблица времени выполнения алгоритмов

Длина строк	Левенштейн	Дамерау	Рекурсивный	Рекурсивный с кэшем
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	2	0
5	0	1	11	0
6	0	1	60	1
7	1	2	327	1
8	1	2	1786	1
9	1	3	9840	2
10	1	2	NaN	2
11	1	2	NaN	2
12	2	3	NaN	3
13	2	3	NaN	3
14	2	4	NaN	4
15	3	4	NaN	5
16	3	5	NaN	5
17	3	5	NaN	6
18	4	6	NaN	7
19	4	7	NaN	8
20	5	7	NaN	8
25	7	11	NaN	13
30	10	16	NaN	19
35	14	21	NaN	27
40	19	28	NaN	36
45	23	35	NaN	45
50	28	43	NaN	56
60	40	61	NaN	81
70	55	83	NaN	115
80	71	107	NaN	151
90	93	133	NaN	192
100	119	66	NaN	237

На рисунке 4.2 представлена зависимость времени обработки строк от размеров строки для всех 4 реализованных алгоритмов. Замеры времени для реализации рекурсивного алгоритма поиска расстояния Дамерау – Левенштейна проводились до длины строк 9, так как его функция очень быстро

растет. На рисунке 4.3 приведены графики без рекурсивного алгоритма Дамерау – Левенштейна, но на более длинных строчках.

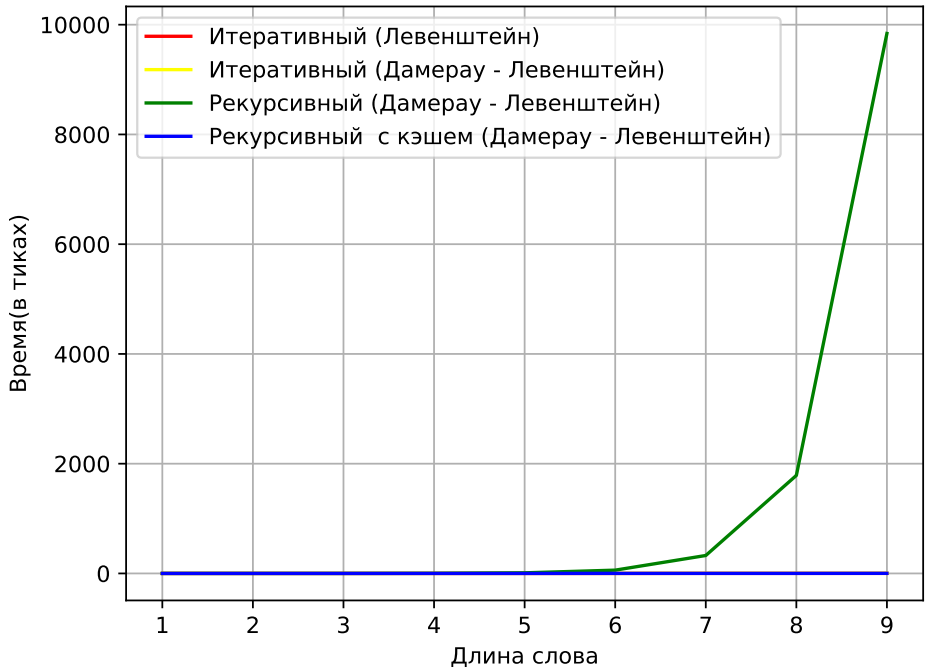

Рисунок 4.2 – Зависимость времени работы реализаций алгоритмов от размера строк для алгоритмов Левенштейна, нерекурсивного Дамерау – Левенштейна, рекурсивного Дамерау – Левенштейна, рекурсивного с кэшированием Дамерау – Левенштейна

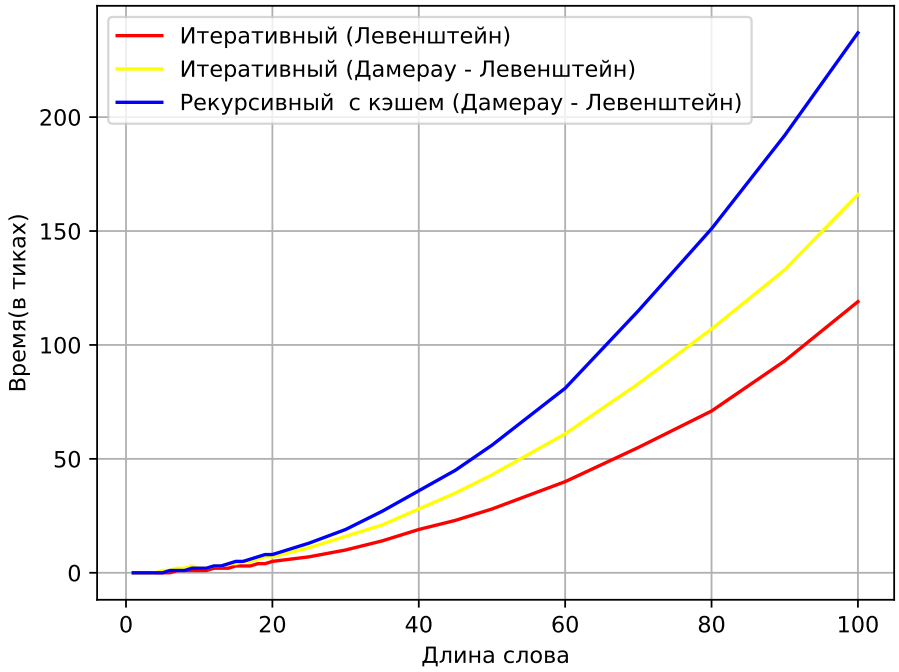

Рисунок 4.3 – Зависимость времени работы реализаций алгоритмов от размера строк для алгоритмов Левенштейна, нерекурсивного Дамерау – Левенштейна, рекурсивного с кэшированием Дамерау – Левенштейна

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:

1) Нерекурсивная реализация алгоритма поиска расстояния Левенштейна:

   — строки S_1, S_2 занимают $(n + m) * sizeof(char)$;

   — длины строк S_1, S_2 занимают $2 * sizeof(int)$;

   — матрица занимает $(m + 1) * (n + 1) * sizeof(int)$;

   — вспомогательные переменные занимают $sizeof(int) + sizeof(bool)$.

   Всего: $(n + m) * sizeof(char) + sizeof(bool) + sizeof(int) * (3 + (m + 1) * (n + 1))$

2) Нерекурсивная реализация алгоритма поиска расстояния Дамерау – Левенштейна:

   — строки S_1, S_2 занимают $(n + m) * sizeof(char)$;

   — длины строк S_1, S_2 занимают $2 * sizeof(int)$;

   — матрица занимает $(m + 1) * (n + 1) * sizeof(int)$;

   — вспомогательные переменные занимают $sizeof(int) + sizeof(bool)$.

   Всего: $(n + m) * sizeof(char) + sizeof(bool) + sizeof(int) * (3 + (m + 1) * (n + 1))$

3) Рекурсивная реализация алгоритма поиска расстояния Дамерау – Левенштейна (для каждого вызова):

   — строки S_1, S_2 занимают $(n + m) * sizeof(char)$;

   — длины строк S_1, S_2 занимают $2 * sizeof(int)$;

   — вспомогательные переменные занимают $sizeof(int) + sizeof(bool)$;

   — адрес возврата – $sizeof(int*)$.

   Для каждого вывода: $(n+m)*sizeof(char)+sizeof(bool)+sizeof(int*)+3 * sizeof(int)$.

   Всего: $(n + m) * ((n + m) * sizeof(char) + sizeof(bool) + sizeof(int*) + 3 * sizeof(int))$.

4) Рекурсивная с кешированием реализация алгоритма поиска расстояния Дамерау – Левенштейна (для каждого вызова):

   — строки S_1, S_2 занимают $(n + m) * sizeof(char)$;

   — длины строк S_1, S_2 занимают $2 * sizeof(int)$;

   — ссылка на матрицу – $sizeof(int*)$;

   — вспомогательные переменные занимают $5 * sizeof(int)$.

   — адрес возврата – $sizeof(int*)$.

   Для каждого вывода: $(n + m) * sizeof(char) + 2 * sizeof(int*) + 7 * sizeof(int)$.

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

Всего: $(n+m)*((n+m)*sizeof(char)+2*sizeof(int*)+7*sizeof(int))+(n+1)*(m+1)*sizeof(int)$

Заключение

В ходе выполнения лабораторной работы поставленная цель была достигнута: были получены навыки динамического программирования на примере задачи поиска редакционных расстояний.

Были изучены и реализованы алгоритмы нахождения расстояния Левенштейна и Дамерау – Левенштейна. Были выполнены замеры затрат реализаций алгоритмов по памяти и по процессорному времени, а также проведены сравнительный анализ двух нерекурсивных алгоритмов и сравнительный анализ алгоритмов поиска расстояния Дамерау – Левенштейна.

В ходе экспериментов было продемонстрировано, что самым затратным по времени является рекурсивный алгоритм Дамерау – Левенштейна, а по памяти – рекурсивный с кешированием алгоритм Дамерау – Левенштейна. Итеративные алгоритмы напротив являются наименее затратными по времени.

Список использованных источников

[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: 20.11.2022).

[3] Linux Mint 21 overview [Электронный ресурс]. Режим доступа: `https://linuxmint.com` (дата обращения: 20.11.2022).