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 о лабораторной работе №1
по дисциплине "Анализ алгоритмов"
на тему "Расстояния Левенштейна и Дамерау-Левенштейна

Студент Коняев Е.А

Группа ИУ7-53Б

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

Дата сдачи отчета_____

Оценка (баллы) _____

Москва — 2022 г.

Оглавление

Введение

Обработка строк является важной частью в сфере программирования и алгоритмики, так как она используется при решении многих важных задач, например:

1) поиска слова в тексте;

2) проверки текстов на орфографию;

3) сравнения файлов.

Среди всех алгоритмов работы со строками одними из важнейших являются методы их сравнения, некоторые из которых будут рассмотрены в данной лабораторной работе.

Глава 1

Аналитическая часть

1.1 Цель и задачи

Целью данной работой является изучение метода динамического программирования на основании определения редакционных расстояний с помощью алгоритмов Левенштейна и Дамерау-Левенштейна. Для достижения поставленной цели требуется решить следующие задачи:

1) изучить и рассмотреть алгоритмы Левенштейна и Дамерау-Левенштейна;
2) построить блок-схемы данных алгоритмов;
3) реализовать каждый из алгоритмов;
4) оценить их ресурсозатратность по памяти;
5) экспериментально оценить временные характеристики алгоритмов;
6) сделать вывод на основании проделанной работы.

1.2 Итерационный алгоритм нахождения расстояния Левенштейна

Расстояние Левенштейна является мерой сходства между двумя строками. Пусть исходная строка S_1, а целевая строка S_2. Тогда рассматриваемое расстояние – это количество **удалений** (Delete), **вставок** (Insert) или **замен** (Replace), которые необходимо совершить для того, чтобы преобразовать S_1 в S_2. Чем больше расстояние Левенштейна, тем больше отличаются строки.

Принято для вышеописанных операций вводить штраф равный единице. Само же расстояние Левенштейна вычисляется по руккурентной формуле (формуле, использующей значения предыдущих членов ряда для вычесления последующих):

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

В итерационном алгоритме нахождения расстояния Левенштейна используется формула (1.1).

1.3 Итерационный алгоритм нахождения расстояния Дамерау-Левенштейна

Алгоритм нахождения расстояния Дамерау-Левенштейна отличается от предыдущего тем, что в нем, помимо операций удаления, вставки, замены, в нем присутствует еще и **транспозиция двух символов** (Match). Данную операцию Домерау предложил из соображений, что перестановка соседних символов местами – одна из самых частых ошибок при наборе рукописного текста. Штраф транспозиции двух символов так же равен единице.

Таким образом, в реккурентной формуле (1.1) необходимо добавить еще одно определение минимума, в случае, когда возможна перестановка двух сосетдних симоволов:

$$D(i,j) = \begin{cases} 0, & \text{i} = 0, \text{j} = 0, \\ i, & \text{j} = 0, \text{i} > 0, \\ j, & \text{i} = 0, \text{j} > 0, \\ \min\{ & \\ \quad D(i,j-1)+1, & \text{i} > 0, \text{j} > 0, \\ \quad D(i-1,j)+1, & S_i = S_{j-1}, \\ \quad D(i-2,j-2)+1, & S_{i-1} = S_j, \\ \quad D(i-1,j-1) + \begin{cases} 0, & \text{если } S_i = S_j, \\ 1, & \text{иначе} \end{cases} & \\ \}, & \\ \min\{ & \\ \quad D(i,j-1)+1, & \\ \quad D(i-1,j)+1, & \text{иначе} \\ \quad D(i-1,j-1) + \begin{cases} 0, & \text{если } S_i = S_j, \\ 1, & \text{иначе} \end{cases} & \\ \} & \end{cases} \tag{1.2}$$

Именно на формуле (1.2) основан итерационный алгоритм нахождения расстояния Дамерау-Левенштейна.

1.4 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна

Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна отличается от его итерационной разновидности тем, что в нем не используется матрица для хранения предыдущих значений, необходимых для подсчета последующих. Вместо этого данные значения вычисляются каждый раз рекурсивно.

1.5 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна с использованием кеша

Данный алгоритм является оптимизацией рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна. Оптимизация заключается в использовании кеша, который представляется собой матрицу, в которую записываются значения, вычисленные на шагах рекурсии. Таким образом, при необходимости вычисления какого-либо нового значения по рекурентной формуле, величины, необходимые для этого не находятся каждый раз: сначала проверяется, было ли вычислено данное значения ранее, и только лишь в случае, если этого не происходило, выполняются рекурсивные вычисления для его получения.

Вывод

В данном разделе были рассмотрены алгоритмы Левенштейна и Дамерау-Левенштейна, позволяющие находить редакционные расстояния для строк.

Глава 2

Конструкторская часть

В данном разделе будут представлены описание типов данных, теоретическая оценка затрат алгоритмов по памяти и схемы алгоритмов.

2.1 Описание типов данных

Для реализации алгоритмов Левенштейна и Дамерау-Левенштейна были использованы типы данных:

1) две строки, каждая из которых является массивом типа *char*;
2) длины строк – тип *int*;
3) структура матрицы, которая состоит из ее размерностей (2 числа типа *int*) и двумерного массива элементов типа *int*.

2.2 Оценка затрат алгоритмов по памяти

Затраты по памяти для алгоритм поиска расстояния Левенштайна (итерационного):

1) длины строк n, m – $2 \cdot sizeof(int)$;
2) матрица – $(n+1) \cdot (m+1) \cdot sizeof(int) + 2 \cdot sizeof(int)$;
3) строки – $(n+m+2) \cdot sizeof(char)$;
4) дополнительные переменные (i, j, res, offset) – $4 \cdot sizeof(int)$;
5) адрес возврата.

Суммарные затраты по памяти:
$(n+1) \cdot (m+1) \cdot sizeof(int) + 7 \cdot sizeof(int) + (n+m+2) \cdot sizeof(char)$

Затраты по памяти для алгоритм поиска расстояния Дамерау-Левенштейна (итерационного):

1) длины строк n, m – $2 \cdot sizeof(int)$;

2) матрица – $(n+1) \cdot (m+1) \cdot sizeof(int) + 2 \cdot sizeof(int)$;

3) строки – $(n+m+2) \cdot sizeof(char)$;

4) дополнительные переменные (i, j, res, offset) – $4 \cdot sizeof(int)$;

5) адрес возврата.

Суммарные затраты по памяти:
$(n+1) \cdot (m+1) \cdot sizeof(int) + 7 \cdot sizeof(int) + (n+m+2) \cdot sizeof(char)$

Затраты по памяти для алгоритм поиска расстояния Дамерау-Левенштейна (рекурсивного) для одного вызова:

1) длины строк n, m – $2 \cdot sizeof(int)$;

2) строки – $(n+m+2) \cdot sizeof(char)$;

3) дополнительные переменные (res, offset) – $2 \cdot sizeof(int)$;

4) адрес возврата.

Суммарные затраты по памяти (R – количество вызовов рекурсии):
$(4 \cdot sizeof(int) + (n+m+2) \cdot sizeof(char)) \cdot R$

Затраты по памяти для алгоритм поиска расстояния Дамерау-Левенштейна (рекурсивного с кешем) для одного вызова:

1) длины строк n, m – $2 \cdot sizeof(int)$;

2) строки – $(n+m+2) \cdot sizeof(char)$;

3) дополнительные переменные (res, offset) – $2 \cdot sizeof(int)$;

4) матрица – $(n+1) \cdot (m+1) \cdot sizeof(int) + 2 \cdot sizeof(int)$;

5) адрес возврата.

Суммарные затраты по памяти (R – количество вызовов рекурсии):
$(6 \cdot sizeof(int) + (n+m+2) \cdot sizeof(char)) \cdot R + (n+1) \cdot (m+1) \cdot sizeof(int)$

2.3 Описания алгоритмов

На рисунках ниже показаны схемы алгоритмов Левенштейна и Дамерау-Левенштейна.

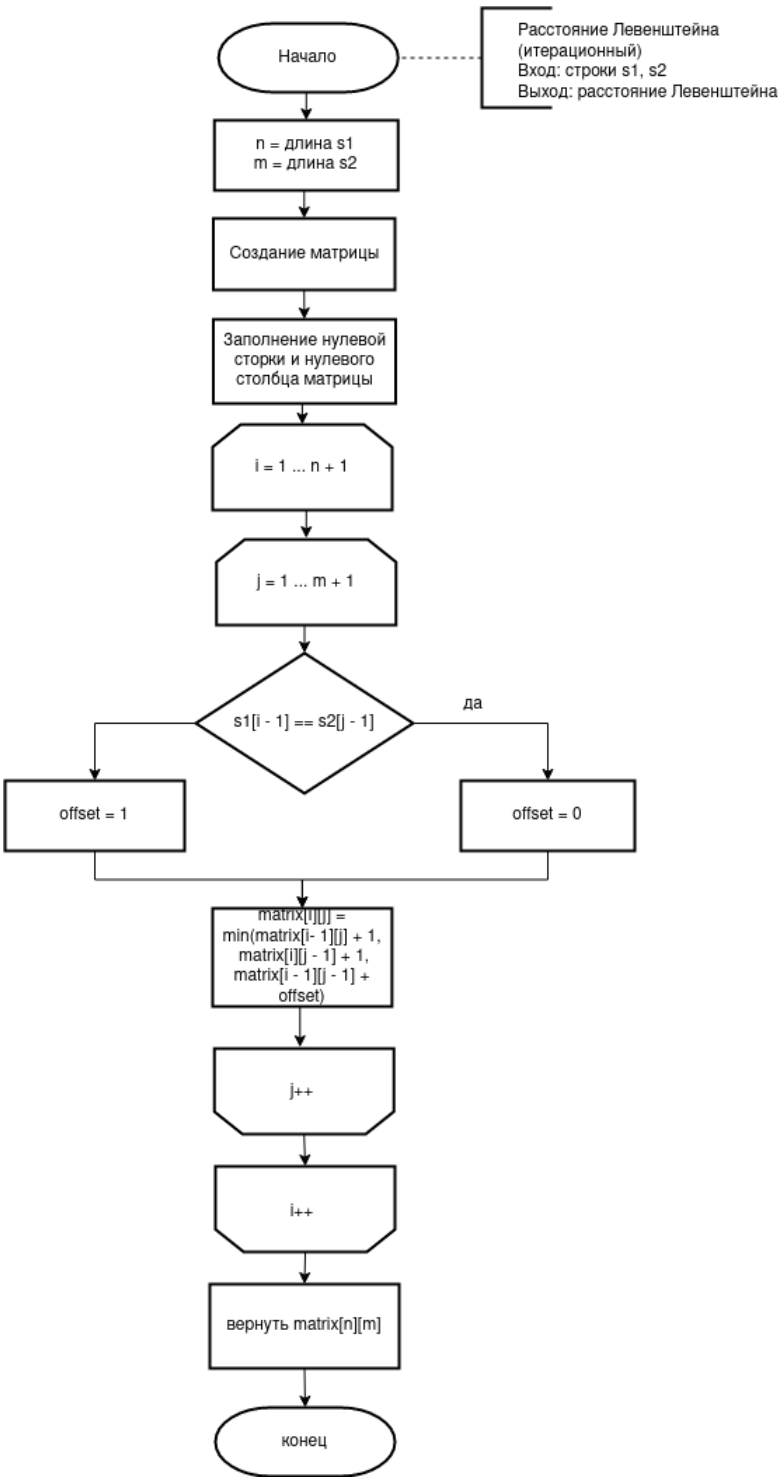

Рисунок 2.1 – Схема итерационного алгоритма поиска расстояния Левенштейна

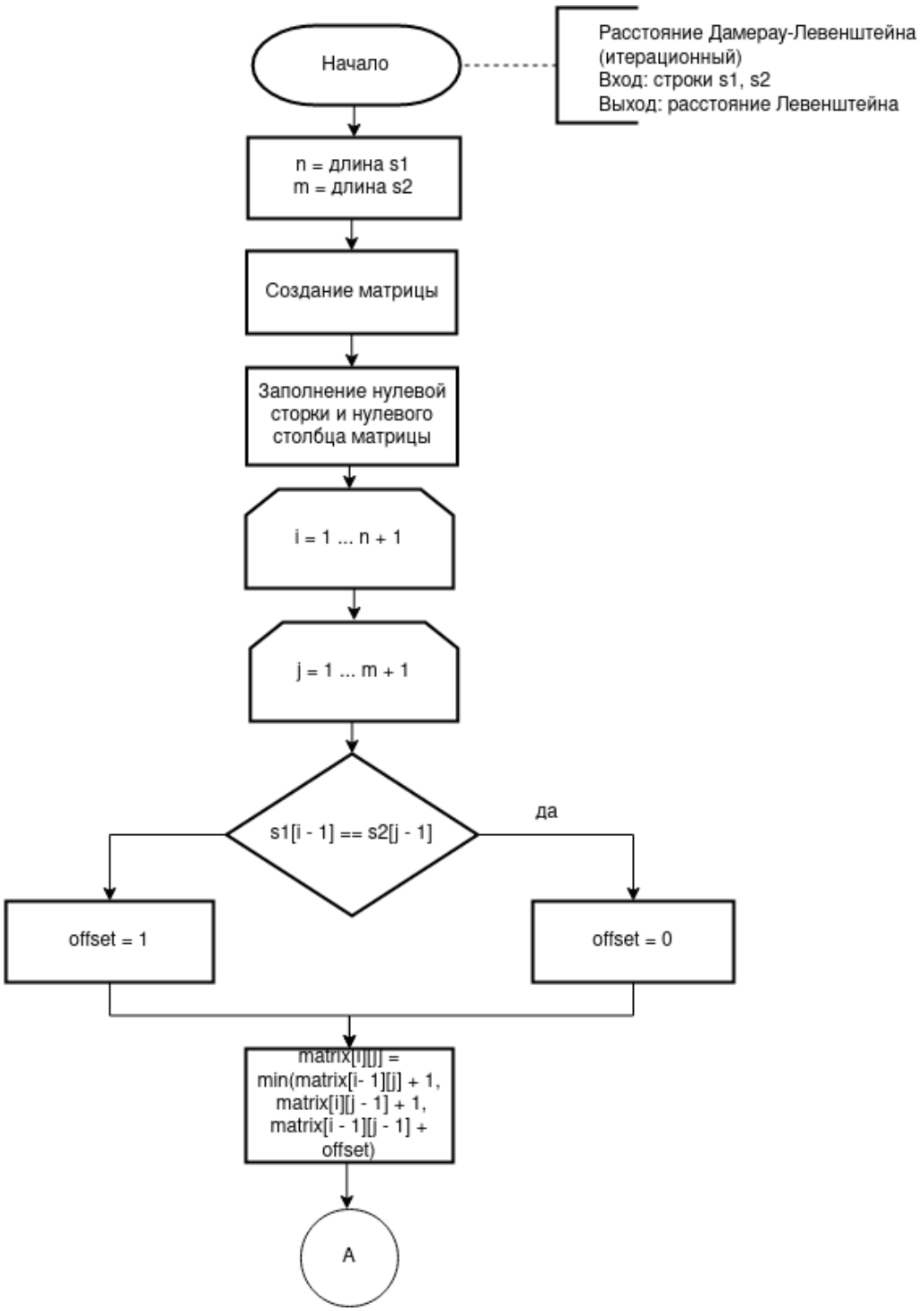

Рисунок 2.2 – Схема итерационного алгоритма поиска расстояния Дамерау-Левенштейна, ч.1

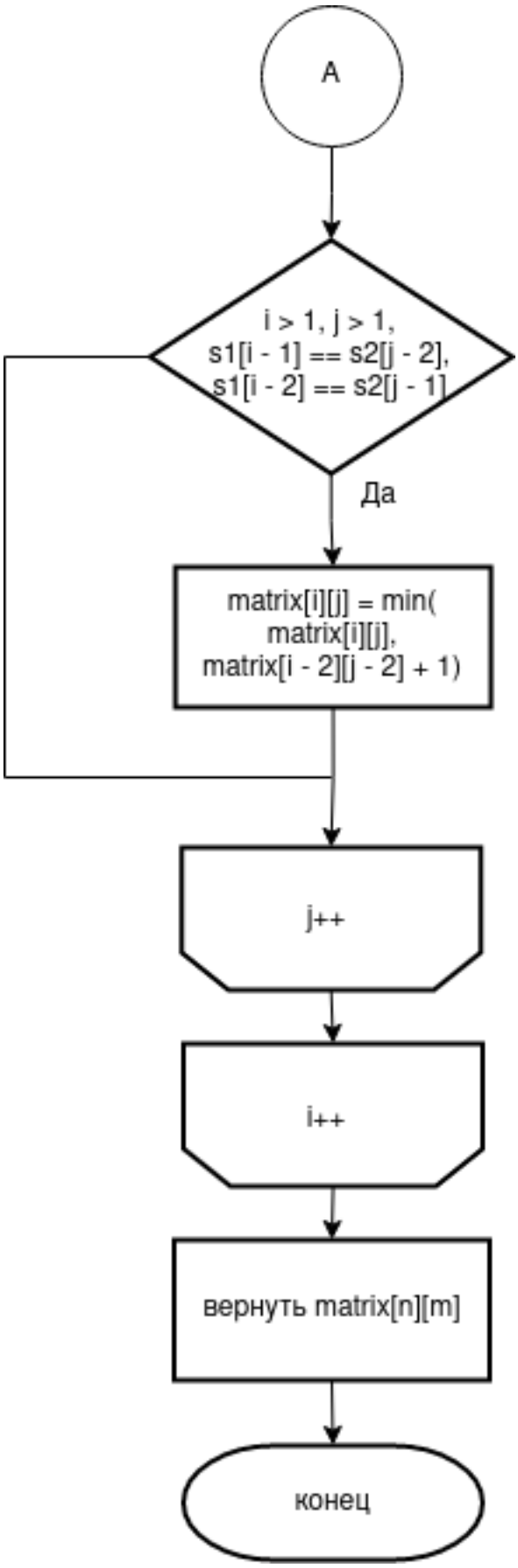

Рисунок 2.3 – Схема итерационного алгоритма поиска расстояния Дамерау-Левенштейна, ч.2

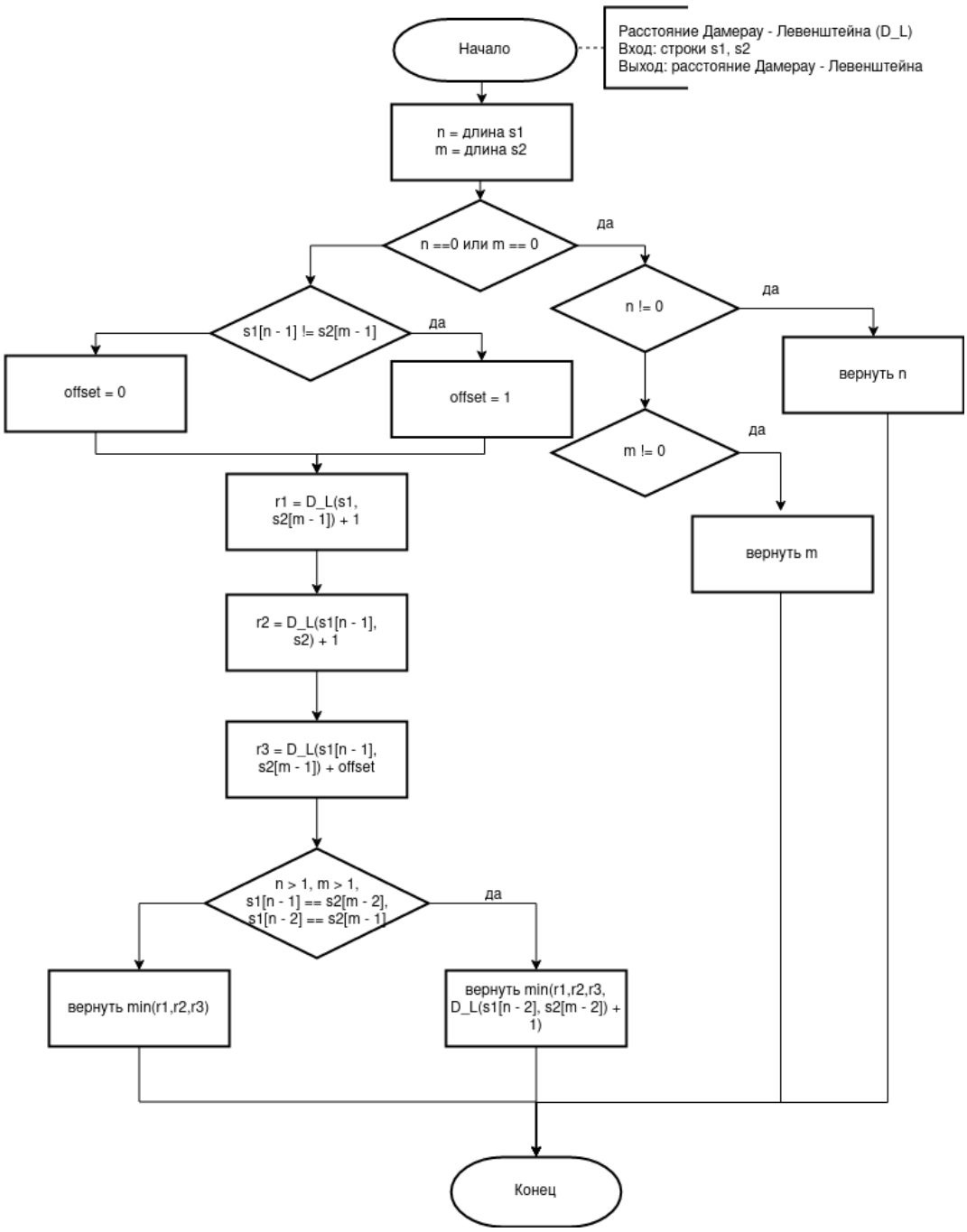

Рисунок 2.4 – Схема рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна

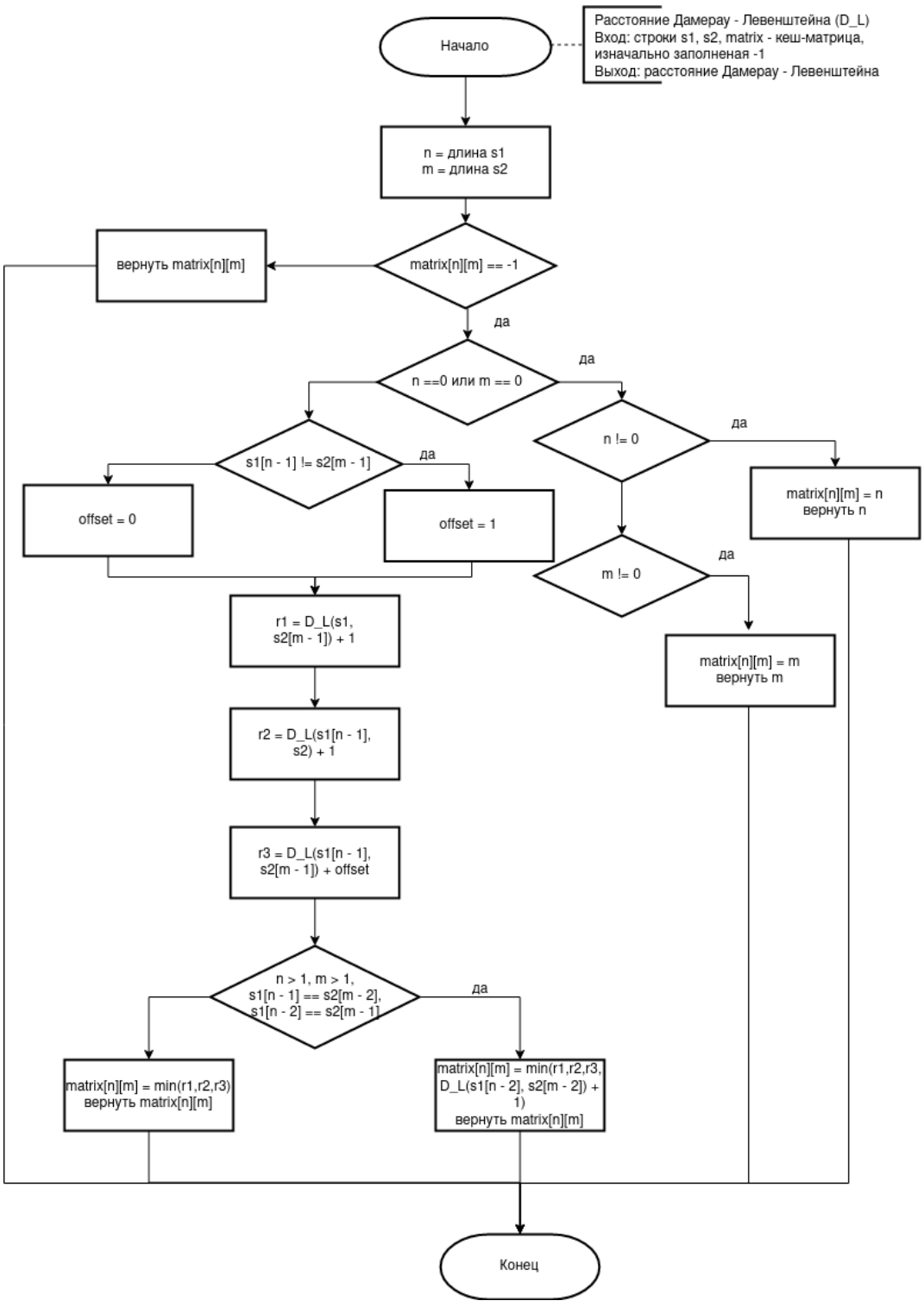

Рисунок 2.5 – Схема рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна с кешем

Вывод

В данном разделе были представлены описания алгоритмов Левенштейна и Дамерау-Левенштейна, а так же проведена теоретическая оценка затрат алгортмов по памяти, которая показала, что рекурсивные алгоритмы менее затратны по памяти.

Глава 3

Технологическая часть

В данном разделе приведены требования к ПО, обоснования выбора языка программирования, среды разработки, приведен способ замера времени выполнения, а также приведены листинги кода.

3.1 Требования к ПО

В программе должна присутствовать возможность:

1) ввода исходных строк, для которых будет находиться расстояния Левенштейна и Дамерау-Левенштейна;

2) поиска искомого расстояния для введеных строк с помощью одного из четырех рассматриваемых алгоритмов;

3) замера процессорного времени выполнения реализаций алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.

3.2 Выбор языка программирования и среды разработки

Для реализации трех алгоритмов сортировок был выбор язык C, так как я уже имею опыт разработки на данном языке программирования.

Средой разработки был выбран CLion. Данный выбор обусловлен тем, что данная среда предоставляет возможность разработки приложений под c/c++ и имеет инструменты для отладки кода.

3.3 Выбор библиотеки и способа для замера времени

Для замера времени выполнения сортировок использовалась стандартная функция библиотеки <time.h> языка C — clock(), которая замеряет процессорное время. Если измерить время перед началом выполнения алгоритма, и после его окончания, то можно получить время выполнения функции. Реализация данной функции приведена в [1].

Поскольку все процессорное время не отдается какой-либо одной задаче (в связи с явлением вытеснения процессов из ядра, квантование процессорного времени), то требуется

усреднить результаты вычислений: замерить совокупное время выполнения реализации алгоритма N раз и вычислить среднее время выполнения.

3.4 Реализации алгоритмов

В листингах 3.1,3.2,3.3,3.4 приведены реализации алгоритмов поиска расстояний Левенштейна (итерационный), Дамера-Левенштейна (итерационный), Дамерау-Левенштейна (рекурсивный), Дамерау-Левенштейна (рекурсивный с кешем) соответсвенно.

Листинг 3.1 – Листинг итерационного алгоритма поиска расстояния Левенштейна

```
1     int dist_lev(char *str_1, char *str_2)
2     {
3         matrix_t *d = create_matrix(strlen(str_1) + 1, strlen(str_2) + 1);
4
5         d->elements[0][0] = 0;
6         for (size_t i = 1; i < d->rows; ++i)
7       d->elements[i][0] = i;
8         for (size_t i = 1; i < d->cols; ++i)
9       d->elements[0][i] = i;
10
11        for (size_t i = 1; i < d->rows; ++i)
12      for (size_t j = 1; j < d->cols; ++j) {
13          int offset = str_1[i - 1] == str_2[j - 1] ? 0 : 1;
14          d->elements[i][j] = get_min(d->elements[i][j - 1] + 1,
15                                      get_min(d->elements[i - 1][j] + 1,
16                                              d->elements[i - 1][j - 1] + offset))
                                                  ;
17      }
18
19        int res = d->elements[d->rows - 1][d->cols - 1];
20        free_matrix(d);
21
22        return res;
23    }
```

Листинг 3.2 – Листинг итерационного алгоритма поиска расстояния Дамерау-Левенштейна

```
1     int dist_dameray_lev(char *str_1, char *str_2)
2     {
3         matrix_t *d = create_matrix(strlen(str_1) + 1, strlen(str_2) + 1);
4
5         d->elements[0][0] = 0;
6         for (size_t i = 1; i < d->rows; ++i)
7       d->elements[i][0] = i;
8         for (size_t i = 1; i < d->cols; ++i)
9       d->elements[0][i] = i;
10
11        for (size_t i = 1; i < d->rows; ++i)
12      for (size_t j = 1; j < d->cols; ++j) {
```

```
13          int offset = str_1[i - 1] == str_2[j - 1] ? 0 : 1;
14          d->elements[i][j] = get_min(d->elements[i][j - 1] + 1,
15                                      get_min(d->elements[i - 1][j] + 1,
16                                              d->elements[i - 1][j - 1] + offset))
                                                ;
17          if (j > 1 && i > 1 && str_1[i - 1] == str_2[j - 2] && str_1[i - 2]
                == str_2[j - 1])
18        d->elements[i][j] = get_min(d->elements[i][j], d->elements[i - 2][j
             - 2] + 1);
19      }
20
21        int res = d->elements[d->rows - 1][d->cols - 1];
22        free_matrix(d);
23
24        return res;
25    }
```

Листинг 3.3 – Листинг рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна

```
1     int dist_dameray_lev_rec(char *str_1, char *str_2, int len_1, int len_2)
2     {
3         if (len_1 == 0)
4       return len_2;
5         if (len_2 == 0)
6       return len_1;
7
8         int offset = str_1[len_1 - 1] == str_2[len_2 - 1] ? 0 : 1;
9         int res = get_min(dist_dameray_lev_rec(str_1, str_2, len_1 - 1,
              len_2) + 1,
10              get_min(dist_dameray_lev_rec(str_1, str_2, len_1, len_2 - 1) +
                    1,
11                      dist_dameray_lev_rec(str_1, str_2, len_1 - 1, len_2 -
                           1) + offset));
12        if (len_1 > 1 && len_2 > 1 && str_1[len_1 - 1] == str_2[len_2 - 2]
              && str_1[len_1 - 2] == str_2[len_2 - 1])
13      res = get_min(res, dist_dameray_lev_rec(str_1, str_2, len_1 - 2, len_2
            - 2) + 1);
14
15        return res;
16    }
```

Листинг 3.4 – Листинг рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна с кешем

```
1     int dist_dameray_lev_rec_hash_call(char *str_1, char *str_2, int len_1,
          int len_2, matrix_t *mat)
2     {
3         if (len_1 == 0)
4       return (mat->elements)[len_1][len_2] = len_2;
5         if (len_2 == 0)
```

```
6         return (mat->elements)[len_1][len_2] = len_1;
7
8           if (mat->elements[len_1 - 1][len_2] == -1)
9         dist_dameray_lev_rec_hash_call(str_1, str_2, len_1 - 1, len_2, mat);
10          if (mat->elements[len_1][len_2 - 1] == -1)
11        dist_dameray_lev_rec_hash_call(str_1, str_2, len_1, len_2 - 1, mat);
12          if (mat->elements[len_1 - 1][len_2 - 1] == -1)
13        dist_dameray_lev_rec_hash_call(str_1, str_2, len_1 - 1, len_2 - 1, mat
              );
14
15          int offset = str_1[len_1 - 1] == str_2[len_2 - 1] ? 0 : 1;
16          mat->elements[len_1][len_2] = get_min(mat->elements[len_1][len_2 -
                1] + 1,
17                        get_min(mat->elements[len_1 - 1][len_2] + 1,
18                                mat->elements[len_1 - 1][len_2 - 1] + offset
                                    ));
19          if (len_1 > 1 && len_2 > 1 && str_1[len_1 - 1] == str_2[len_2 - 2]
              && str_1[len_1 - 2] == str_2[len_2 - 1]) {
20        if (mat->elements[len_1 - 2][len_2 - 2] == -1)
21          dist_dameray_lev_rec_hash_call(str_1, str_2, len_1 - 2, len_2 - 2,
                mat);
22        mat->elements[len_1][len_2] = get_min(mat->elements[len_1][len_2], mat
            ->elements[len_1 - 2][len_2 - 2] + 1);
23          }
24
25          return mat->elements[len_1][len_2];
26      }
27
28      int dist_dameray_lev_rec_hash(char *str_1, char *str_2)
29      {
30          matrix_t *d = create_matrix(strlen(str_1) + 1, strlen(str_2) + 1);
31          clear_matrix_elems(d);
32
33          int res = dist_dameray_lev_rec_hash_call(str_1, str_2, strlen(str_1)
                , strlen(str_2), d);
34          free_matrix(d);
35
36          return res;
37      }
```

3.5 Тестирование алгоритмов

В таблице 3.1 приведены тесты для функций, реализующих алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна. Все тесты пройдены успешно.

Таблица 3.1 – : Тестирование функций

Первая строка	Вторая строка	Результат (Лев.)	Результат (Дам.-Лев.)
"пустая строка"	"пустая строка"	0	0
fit	"пустая строка"	3	3
fit	fit	0	0
fit	$fiting$	3	3
$class$	$calss$	2	1

Вывод

В данном разделе были разработаны исходные коды алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.

Глава 4

Экспериментальная часть

4.1 Технические характеристики

Ниже приведены технические характеристики устройства, на котором было проведено тестирование ПО:

1) операционная система Windows-10, 64-bit;
2) оперативная память 16 ГБ;
3) процессор Intel(R) Core(TM) i7-8565U CPU @ 1.80 Ггц, 4 ядра, 8 логических процессоров.

4.2 Замеры времени

Введем обозначения: Lev – итерационный алгоритм поиска расстояния Левенштейна, DamLev – итерационный алгоритм поиска расстояния Дамерау-Левенштейна, DamLevRec – рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна, DamLevRecHash – итерационный алгоритм поиска расстояния Дамерау-Левенштейна с кешем.

В таблице 4.1 приведены результаты замеров времени алгоритмов для входных строк разной длины.

Таблица 4.1 – Таблица замера времени выполнения алгоритмов на строках, имеющих разные длины

Длина стоки	Lev	DamLev	DamLevRec	DamLevRecHash
1	0.2	0.16	0.14	0.32
2	0.3	0.32	0.24	0.62
3	0.56	0.68	1.12	1.0
4	0.76	1.0	4.26	1.3
5	1.02	1.3	21.7	1.72
6	1.05	1.6	127.28	2.18
7	1.1	2.0	567.96	2.3
8	1.24	2.3	3170.12	3.08
9	1.42	2.3	18337.36	3.64
10	1.94	2.38	102172.82	4.2

Зависимость времени работы алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна от длины входных строк представлена на рис. 4.1.

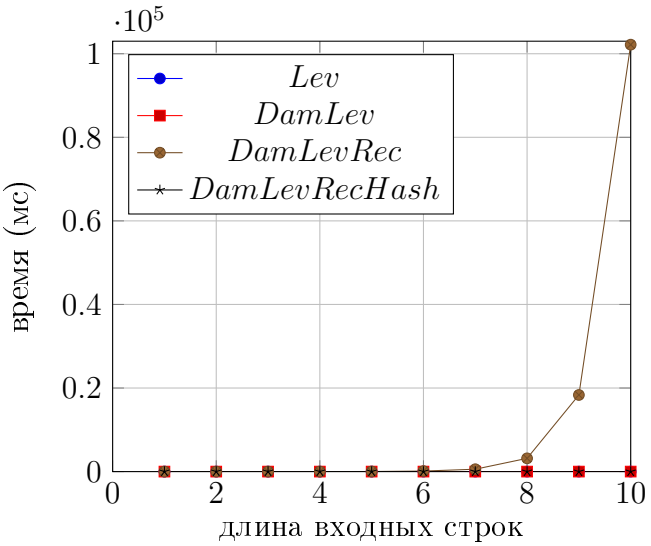

Рисунок 4.1 – Зависимость времени от длины входных строк

Зависимость времени работы итерационных алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна от длины входных строк представлена на рис. 4.2.

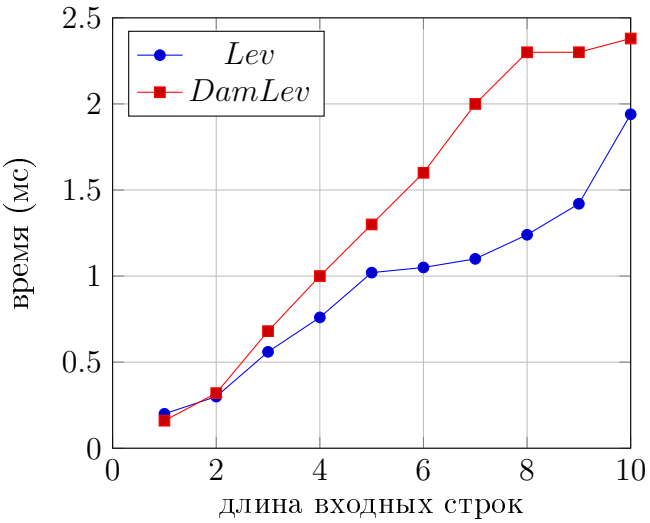

Рисунок 4.2 – Зависимость времени от длины входных строк

Вывод

Результаты замеров показали, что рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна работает дольше всего. При этом его оптимизация с кешем – работает в разы быстрее. Итерационные алгоритмы нахождения расстояний Левенштейна и Дамерау-Левенштейна оказались наиболее быстрыми, причем на длине строки от 1 до 10 элементов итерационный алгоритм Левенштейна в среднем работает немного быстрее, нежели итерационный алгоритм нахождения расстояния Дамера-Левенштейна.

Заключение

Цель лабораторной работы достигнута – был изучен метод динамического программирования на основании определения редакционных расстояний с помощью алгоритмов Левенштейна и Дамерау-Левенштейна. Все задачи решены:

1) были изучены и рассмотрены алгоритмы Левенштейна и Дамерау-Левенштейна;
2) были построены блок-схемы выбранных алгоритмов;
3) был реализован каждый из алгоритмов;
4) была рассчитана их трудоемкость по памяти;
5) были экспериментально получены временные характеристики алгоритмов;
6) были сделаны выводы на основании проделанной работы

На основании проведенных экспериментов было определено, что время работы алгоритмов увеличивается в геометрической прогрессии в зависимости от длины входных строк. Самым медленным по скорости выполнения, но наименне затратным по памяти оказался рекурсивный алгоритм определения расстояния Дамерау-Левенштейна. Самым быстрым же оказался итерационный алгоритм нахождения расстояния Левенштейна, который одновременно с этим, оказался одним из самых затратных по памяти, так как в его реализации дополнительно используется матрица, размером, соответсвующим длине входных строк.

Список использованных источников

[1] The Open Group Base Time [Электронный ресурс]. - URL: https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/time.h.html (дата обращения: 21.10.2022).

[2] Погорелов, Д. А. Сравнительный анализ алгоритмов редакционного расстояния Левенштейна и Дамерау-Левенштейна / Д. А. Погорелов, А. М. Тарзанов. - Синергия Наук. - 2019. URL: https://elibrary.ru/item.asp?id=36907767 (дата обращения: 21.10.2022).

[3] Ингерсолл, Г. С. Обработка неструктурированных текстов / Г. С. Ингерсолл, Т. С. Мортон, Э. Л. Фэррис. - ЛитРес, 2022.