МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ИМ. Н. Э. БАУМАНА

УДК № 004.822
№ госрегистрации _____
Инв. №_____

УТВЕРЖДАЮ

Преподаватель

_____

«_____» _____ 2019 г.

ОТЧЁТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ №1

Редакционное расстояние
(промежуточный)

Студент _____ А. А. Куприй

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

Москва 2019

СПИСОК ИЛЛЮСТРАЦИЙ

РЕФЕРАТ

Отчет содержит 29 стр., 10 рис., 4 табл..

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Для выполнения данной лабораторной работы необходимо решить следующие задачи:

— Изучение расстояний Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками

— Получение практических навыков реализации указанных алгоритмов: двух алгоритмов в матричной версии и оного из алгоритмов в рекурсивной версии

— Сравнительный анализ линейной и рекурсивной реализации выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам (времени и памяти)

— Экспериментальное подтверждение различий во временой эффективности рекурсивной и нерекурсивной реализации выбранного алгоритма определения расстояния межлу строками при помощи разработанного программного обеспечения на материале замеров процессорного времен выполнения реализации на варьирующихся длинах строк.

1 Аналитический раздел

1.0.1 Расстояния Левенштейна

Расстояние Левенштейна между двумя строками - это минимальная сумма произведений количества операций вставки, удаления и замены одного символа, необходимых для превращения одной строки в другую, на их стоимость.

Вышеописанные операции имеют следующие обозначения:

— I $(insert)$ - вставка;

— D $(delete)$ - удаление;

— R $(replace)$ - замена;

— M $(match)$ - совпадение;

При этом $cost(x)$ есть обозначение стоимости некоторой операции x. Будем считать, что символы в строках нумеруются с первого. Пусть S_1 и S_2 - две строки с длинами N и M соответственно. Тогда расстояние Левенштейна D(M, N) вычисляется по формуле (1.1):

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i * cost(D), & j = 0, i > 0 \\ j * cost(I), & i = 0, j > 0 \\ min(& \\ D(i,j-1) + cost(I), & \\ D(i-1,j) + cost(D), & j > 0, i > 0 \\ D(i-1,j-1) + mrcost(S_1[i], S_2[j]) & \\ ) & \end{cases} \tag{1.1}$$

где $min(a, b, c)$ возвращает наименьшее значение из a, b, c; а $mrcost(x_1, x_2)$ - 0, если символы x_1, x_2 совпадают, и $cost(R)$ иначе.

1.0.2 Расстояние Дамерау-Левенштейна

Определение расстояния Дамерау-Левенштейна аналогично определению расстояния Левенштейна с учётом новой операции - перестановки соседних символов (транспозиции). Соответственно, обозначения операций:

— I ($insert$) - вставка;

— D ($delete$) - удаление;

— R ($replace$) - замена;

— T ($transpose$) - перестановка соседних символов.

— M ($match$) - совпадение;

При тех же обозначениях имеем формулы (1.2) и (1.3):

$$D(i,j) = \begin{cases} min(A, D(i-2, j-2) + cost(T), & i > 1, j > 1, \\ & S_1[i] = S_2[j-1], \\ & S_1[i-1] = S_2[j] \\ A & \text{Иначе} \end{cases} \tag{1.2}$$

где A:

$$A = \begin{cases} 0, & i = 0, j = 0 \\ i * cost(D), & j = 0, i > 0 \\ j * cost(I), & i = 0, j > 0 \\ min(& \\ D(i,j-1) + cost(I), & \\ D(i-1, j) + cost(D), & j > 0, i > 0 \\ D(i-1, j-1) + mrcost(S_1[i], S_2[j]) & \\ ) & \end{cases} \tag{1.3}$$

2 Конструкторский раздел

В данном разделе будет произведена конкретизация задач и проанализированы алгоритмы.

2.1 IDEF0 Модель

На рисунке 2.1 приведена IDEF0 функциональная модель вычисления редакционного расстояния.

Рисунок 2.1 — IDEF0 функциональная модель

2.2 Разработка алгоритмов

2.2.1 Алгоритм Вагнера-Фишера

Алгоритм нахождения расстояния Вагнера-Фишера - это матричная реализация поиска расстояния Левенштейна, схема данного алгоритма приведена на рисунке 2.2.

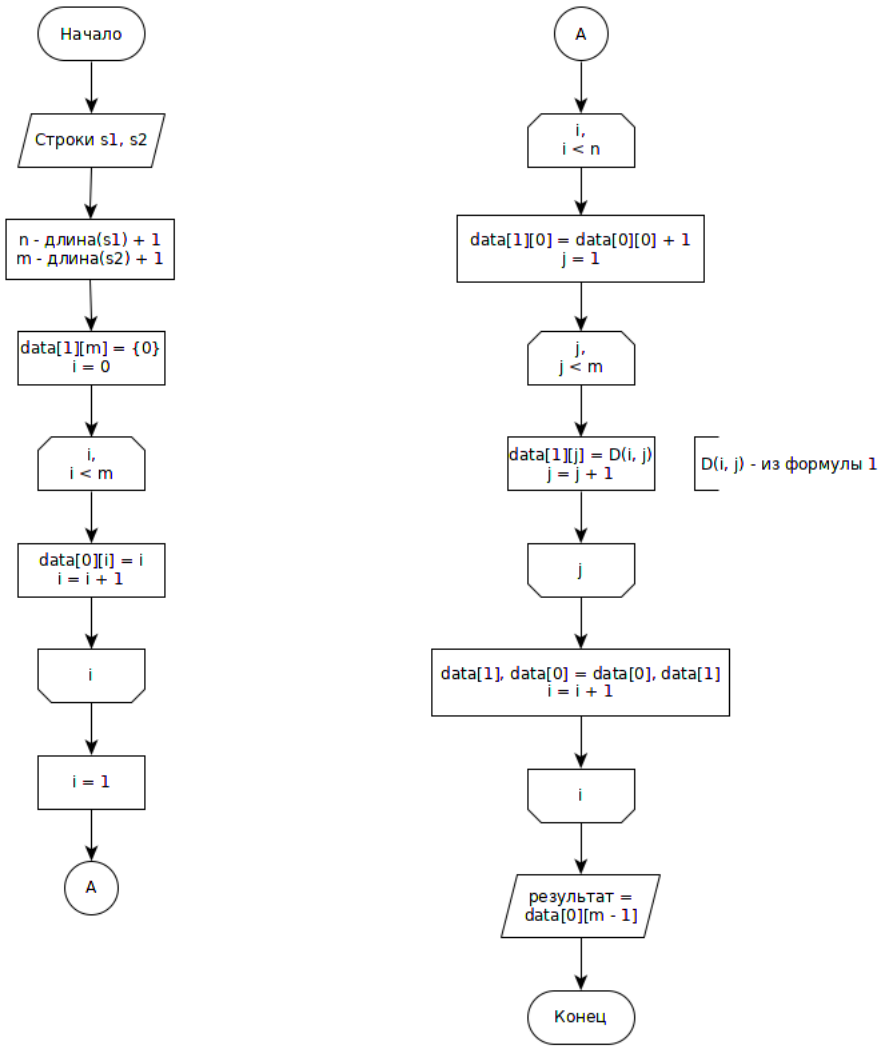

Рисунок 2.2 — Алгоритм Вагнера-Фишера

2.2.2 Матричный алгоритм Дамерау-Левенштайна

Матричный алгоритм нахождения расстояния Дамерау-Левенштайна - это модификация алгоритма Вагнера-Фишера, в котором добавлена операция транспозиции, схема данного алгоритма приведена на рисунке 2.3.

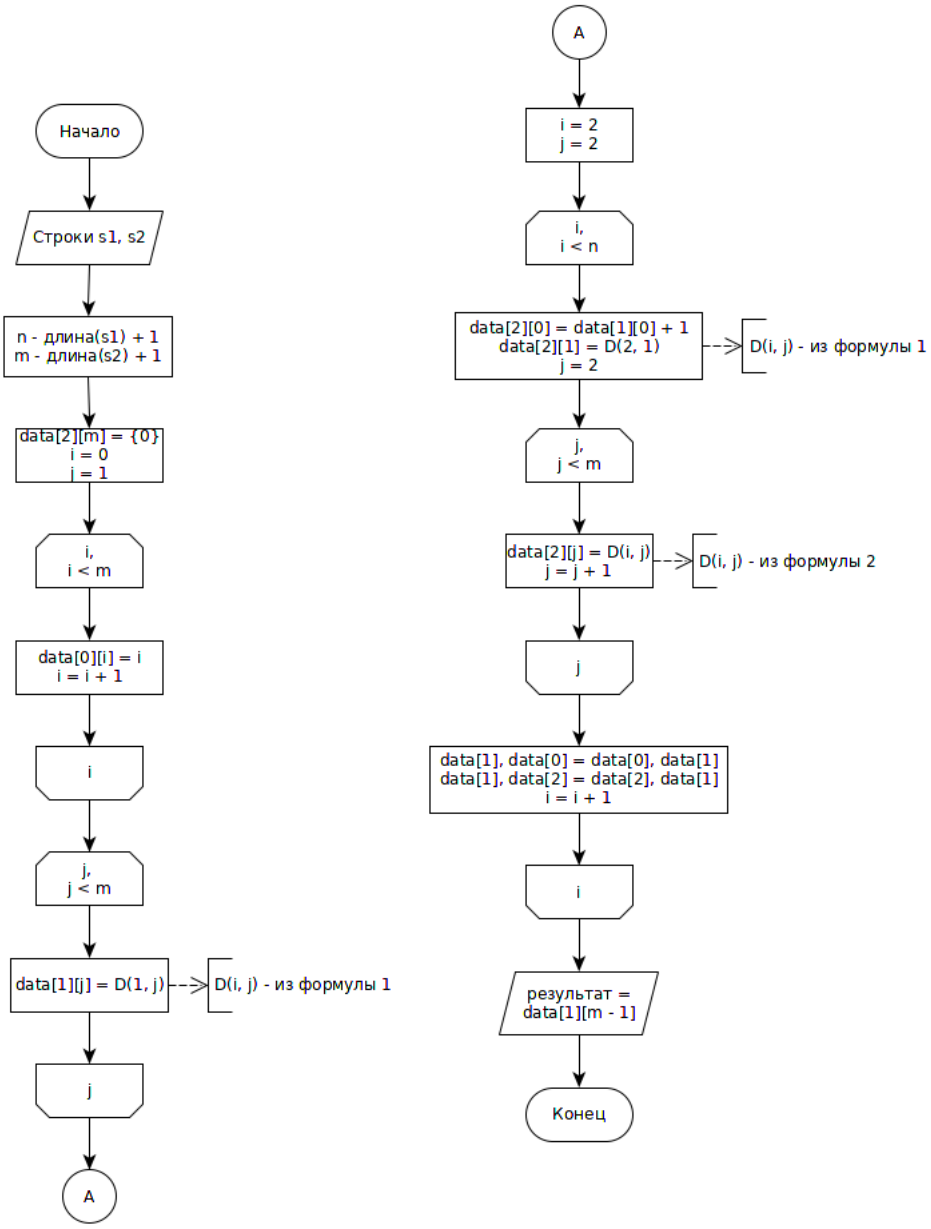

Рисунок 2.3 — Матричный алгоритм Дамерау-Левенштайна

2.2.3 Рекурсивный алгоритм Дамерау-Левенштайна

В рекурсивном алгоритме нахождения расстояния Дамерау-Левенштайна происходит поиск редакционного расстояния до тех пор, пока длина хотя

бы одного из слов не равна 0, схема данного алгоритма приведена на рисунке 2.4

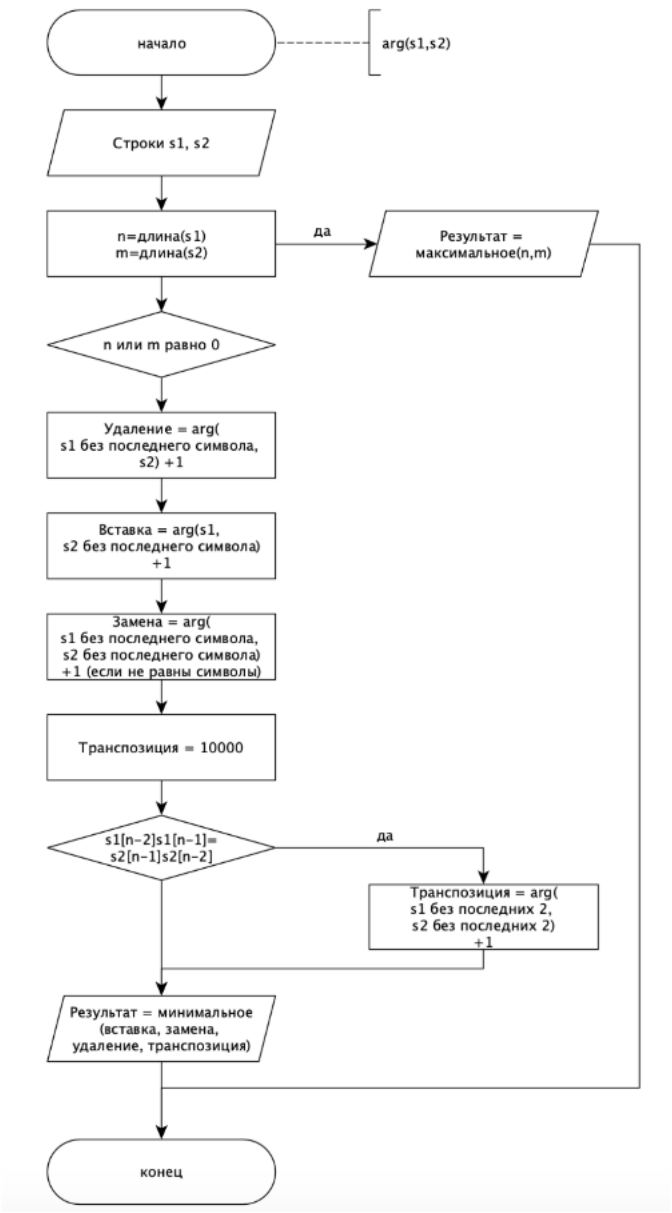

Рисунок 2.4 — Рекурсивный алгоритм Дамерау-Левенштайна

2.3 Сравнительный анализ алгоритмов

2.3.1 Сравнение

Матричная реализация имеет сложность $\Omega(mn)$, где m и n - это длины строк. В самом коротком пути рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна имеет сложность $\Omega(4^{min(m,n)})$, а максимальная сложность $\Omega(4^{m+n+1})$, где m и n - длины строк.

2.3.2 Вывод

На основе вышеприведенного анализа можно сделать вывод о том, что матричная реализация быстрее рекурсивной при больших значениях длин строк.

3 Технологический раздел

В данном разделе приводятся описания требований к программному обеспечению, средства реализации, листинги кода и описания тестирования.

3.1 Требования к программному обеспечению

Требования к вводу:

— на вход подаются два слова;

— каждое слово завершается символом переноса строки;

— пустое слово допускается.

Требования к выводу:

— редакционное расстояние;

3.2 Средства реализации

В качестве языка программирования мною был выбран C++.

Для измерения времени использовалась встроенная библиотека Chrono.

3.3 Листинги кода

В листингах 3.1 - 3.3 приведена реализация описанных алгоритмов.

Листинг 3.1 — Расстояние Левенштейна (матричная реализация)

```
1  int levensteinDistance(char *str1, char *str2)
2  {
3      unsigned lenStr1 = strlen(str1);
4      unsigned lenStr2 = strlen(str2);
5      unsigned rows, columns;
6      bool isNotSame;
7      int result;
8
9      if (lenStr1 == 0 || lenStr2 == 0)
10     {
11         return (lenStr1 > lenStr2) ? lenStr1 : lenStr2;
12     }
```

```
13
14      rows = lenStr1 + 1;
15      columns = lenStr2 + 1;
16
17      int *data[2];
18      data[0] = new int[columns];
19      data[1] = new int[columns];
20
21      for (unsigned i = 0; i < columns; i++)
22      {
23          data[0][i] = i;
24      }
25
26      for (unsigned i = 1; i < rows; i++)
27      {
28          data[1][0] = data[0][0] + 1;
29
30          for (unsigned j = 1; j < columns; j++)
31          {
32              isNotSame = (str1[i - 1] != str2[j - 1]) ? 1 : 0;
33
34              data[1][j] = my_min(data[1][j - 1] + 1, data[0][j] + 1,
                    data[0][j - 1] + isNotSame);
35
36              cout << data[1][j] << " ";
37          }
38
39          cout << endl;
40
41          std::swap(data[1], data[0]);
42      }
43
44      result = data[0][columns - 1];
45
46      delete data[0];
47      delete data[1];
48
49      return result;
50  }
```

Листинг 3.2 — Расстояние Дамерау-Левенштейна (матричная реализация)

```
1  int levensteinDistanceTransposition(char *str1, char *str2)
2  {
3      unsigned lenStr1 = strlen(str1);
4      unsigned lenStr2 = strlen(str2);
5      unsigned rows, columns;
```

```
6       bool isNotSameTrans;
7       bool isNotSame;
8       int result;
9
10      if (lenStr1 == 0 || lenStr2 == 0)
11      {
12          return (lenStr1 > lenStr2) ? lenStr1 : lenStr2;
13      }
14
15      if (lenStr1 < 2 || lenStr2 < 2)
16      {
17          return levensteinDistance(str1, str2);
18      }
19
20      rows = lenStr1 + 1;
21      columns = lenStr2 + 1;
22
23      int *data[3];
24      data[0] = new int[columns];
25      data[1] = new int[columns];
26      data[2] = new int[columns];
27
28      for (unsigned i = 0; i < columns; i++)
29      {
30          data[0][i] = i;
31      }
32
33      data[1][0] = 1;
34
35      for (unsigned j = 1; j < columns; j++)
36      {
37          isNotSame = (str1[0] != str2[j - 1]) ? 1 : 0;
38
39          data[1][j] = my_min(data[1][j - 1] + 1, data[0][j] + 1, data[0][j -
                1] + isNotSame);
40      }
41
42      for (unsigned i = 2; i < rows; i++)
43      {
44          data[2][0] = data[1][0] + 1;
45
46          isNotSame = (str1[i - 1] != str2[0]) ? 1 : 0;
47
48          data[2][1] = my_min(data[1][1] + 1, data[2][0] + 1, data[1][0] +
                isNotSame);
49
```

```
50          for (unsigned j = 2; j < columns; j++)
51          {
52              //isNotSameTrans = ((str1[i - 1] != str2[j - 2]) && (str1[i -2]
                    != str2[j - 1])) ? 1 : 0 ;
53              isNotSame = (str1[i - 1] != str2[j - 1]) ? 1 : 0;
54
55              if (((str1[i - 1] == str2[j - 2]) && (str1[i -2] == str2[j -
                    1])) ? 1 : 0)
56              {
57                  data[2][j] = my_min(data[2][j - 1] + 1, data[1][j] + 1,
                        data[1][j - 1] + isNotSame, data[0][j - 2] + 1);
58              }
59              else
60              {
61                  data[2][j] = my_min(data[2][j - 1] + 1, data[1][j] + 1,
                        data[1][j - 1] + isNotSame);
62              }
63          }
64
65          for (int j = 1; j < columns; j++)
66          {
67              cout << data[1][j] << " ";
68          }
69
70          cout << endl;
71
72          std::swap(data[0], data[1]);
73          std::swap(data[1], data[2]);
74      }
75
76      for (int i = 1; i < columns; i++)
77      {
78          cout << data[1][i] << " ";
79      }
80
81      cout << endl;
82
83      result = data[1][columns - 1];
84
85      delete data[0];
86      delete data[1];
87      delete data[2];
88
89      return result;
90  }
```

Листинг 3.3 — Расстояние Левенштейна (рекурсивная Дамерау-Левенштейна реализация)

```
1  int levensteinRecursiveDistance(char *str1, int lenStr1, char *str2, int
       lenStr2)
2  {
3      bool isNotSame;
4      int result;
5
6      if (lenStr1 == 0)
7      {
8          return lenStr2;
9      }
10     if (lenStr2 == 0)
11     {
12         return lenStr1;
13     }
14
15     isNotSame = str1[lenStr1 - 1] != str2[lenStr2 - 1];
16
17     result = my_min(levensteinRecursiveDistance(str1, lenStr1 - 1, str2,
           lenStr2) + 1, levensteinRecursiveDistance(str1, lenStr1, str2,
           lenStr2 - 1),
18             levensteinRecursiveDistance(str1, lenStr1 - 1, str2, lenStr2 -
                   1) + isNotSame);
19
20     if (lenStr1 > 1 && lenStr2 > 1)
21     {
22         if (str1[lenStr1 - 1] == str2[lenStr2 - 2] && str1[lenStr1 - 2] ==
               str2[lenStr2 - 1])
23         {
24             result = std::min(result, levensteinRecursiveDistance(str1,
                   lenStr1 - 2, str2, lenStr2 - 2) + 1);
25         }
26     }
27
28     return result;
29 }
```

3.4 Описание тестирования

Для тестирования программы были подготовлены данные, представленные в таблице 3.1.

Таблица 3.1 — Тестовые данные

№	Строка 1	Строка 2	Ожидаемое расстояние Левенштейна	Ожидаемое расстояние Дамерау-Левенштейна
1	some	any	4	4
2		nothing	7	7
3			0	0
4	bashrc	bashcr	2	1
5	bus	BuS	2	2
6	electricity	city	7	7
7	powerful	powerless	4	4
8	grow	flow	2	2
9	rise	rice	1	1
10	legal	illegal	2	2
11	same	same	0	0

3.5 Результаты тестирования

Тестирование всех трёх реализаций алгоритмов прошло успешно. Результаты тестов представлены в таблицах 4.1, 4.2, 4.3.

Таблица 3.2 — Результаты тестирования алгоритма Вагнера-Фишера

№	Строка 1	Строка 2	Расстояние Левенштейна	Ожидаемое расстояние Левенштейна
1	some	any	4	4
2		nothing	7	7
3			0	0
4	bashrc	bashcr	2	2
5	bus	BuS	2	2
6	electricity	city	7	7
7	powerful	powerless	4	4
8	grow	flow	2	2
9	rise	rice	1	1
10	legal	illegal	2	2
11	same	same	0	0

Таблица 3.3 — Результаты тестирования рекурсивного алгоритма Дамерау-Левенштейна

№	Строка 1	Строка 2	Расстояние Дамерау-Левенштейна	Ожидаемое расстояние Дамерау-Левенштейна
1	some	any	4	4
2		nothing	7	7
3			0	0
4	bashrc	bashcr	1	1
5	bus	BuS	2	2
6	electricity	city	7	7
7	powerful	powerless	4	4
8	grow	flow	2	2
9	rise	rice	1	1
10	legal	illegal	2	2
11	same	same	0	0

Таблица 3.4 — Результаты тестирования рекурсивного алгоритма Дамерау-Левенштейна

№	Строка 1	Строка 2	Расстояние Дамерау-Левенштейна	Ожидаемое расстояние Дамерау-Левенштейна
1	some	any	4	4
2		nothing	7	7
3			0	0
4	bashrc	bashcr	1	1
5	bus	BuS	2	2
6	electricity	city	7	7
7	powerful	powerless	4	4
8	grow	flow	2	2
9	rise	rice	1	1
10	legal	illegal	2	2
11	same	same	0	0

Все тесты были успешно пройдены.

3.6 Вывод

Были сформулированы требования к ПО, выбраны средства реализации и подготовлены тестовые данные.

4 Исследовательский раздел

В данном разделе привидены и проанализированы примеры работы программы редакционного расстояния.

4.1 Примеры работы

На рисунках 4.1, 4.2, 4.3, 4.4 показана работа программы с различными входными данными.

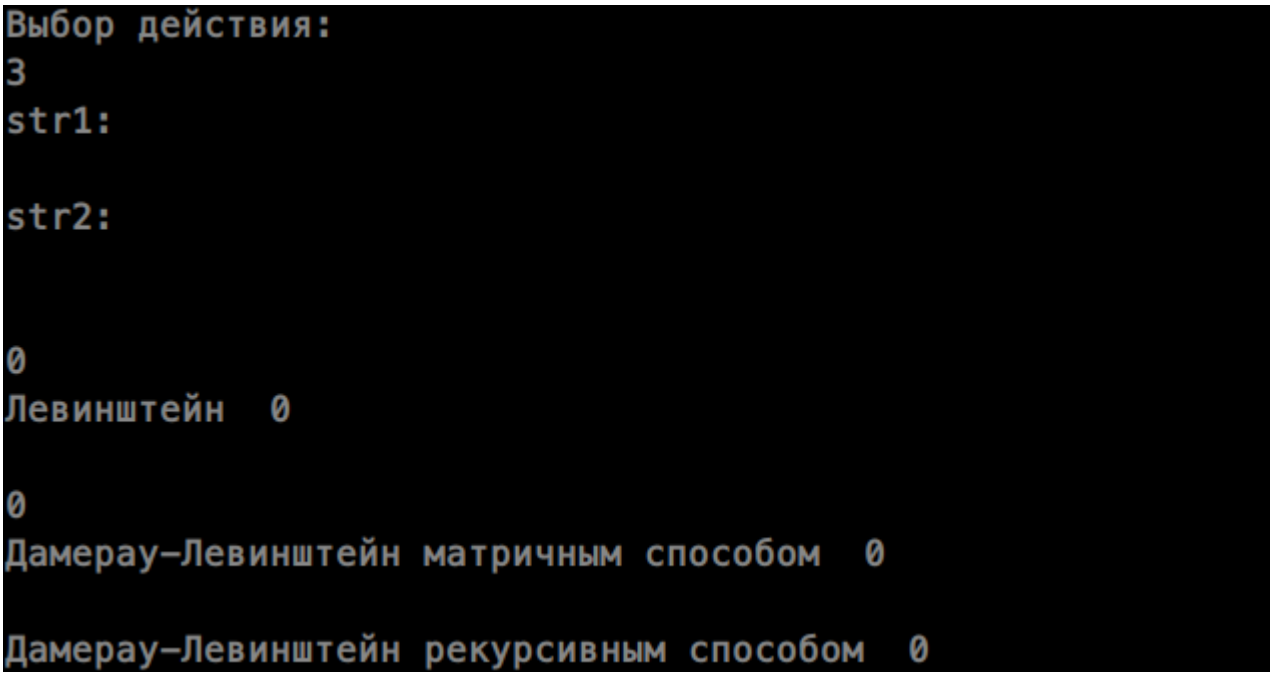

Рисунок 4.1 — Пустое слово

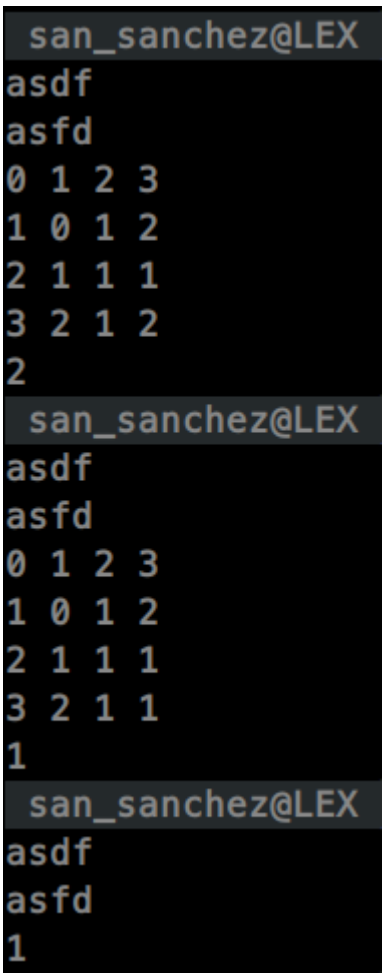

Рисунок 4.2 — Транспозиция

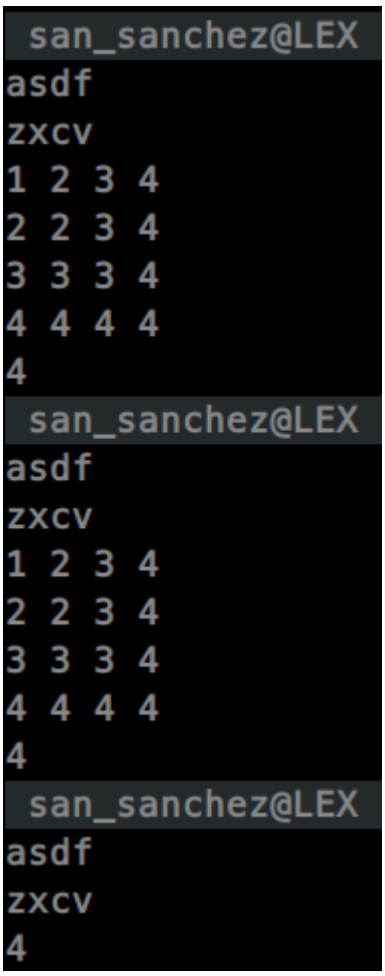

Рисунок 4.3 — Разные слова

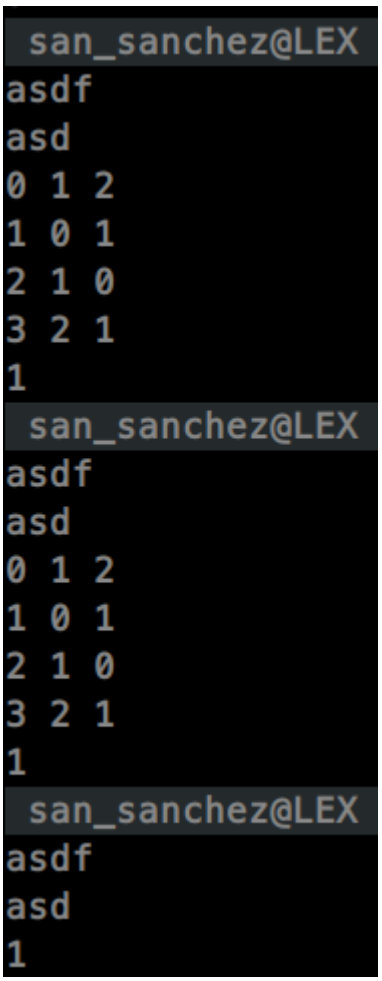

Рисунок 4.4 — Пропущена одна буква

4.2 Эксперименты по замеру времени

Чтобы подтвердить вывод об оценке сложности алгоритмов поиска редакционного расстояния, проведём эксперименты по замеру времени и построим графики зависимости времени выполнения данных алгоритмов от длины обрабатываемых слов.

4.2.1 Эксперимент 1

На рисунке 4.5 приведён график сравнения алгоритма Вагнера-Фишера и матричного алгоритма Дамерау-Левенштейна. Для этого эксперимента было сгенерировано 100 пар полностью не совпадающих строк с диапазоном длин от 10 до 1000. Как видно, законы изменения времени выполнения этих алго-

ритмов практически одинаковы и отличаются лишь на некоторый постоянный коэффициент.

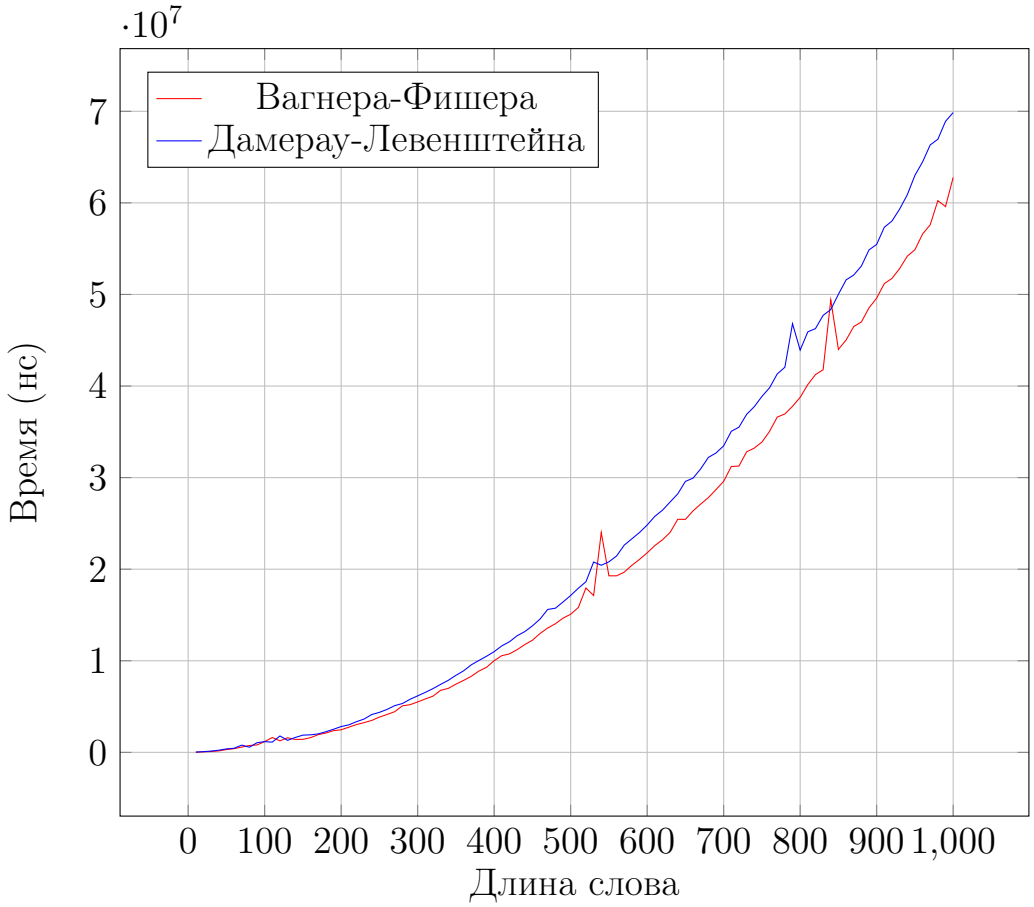

Рисунок 4.5 — График сравнения алгоритма Вагнера-Фишера и матричного алгоритма Дамерау-Левенштейна

4.2.2 Эксперимент 2

На рисунке 4.6 приведён график сравнения рекурсивного и матричного алгоритмов нахождения расстояния Дамерау-Левенштейна. Для этого эксперимента было сгенерировано 10 пар полностью различных слов с диапазоном длин от 1 до 10. Количество времени, необходимого для выполнения рекурсивного алгоритма, растёт экспоненциально, в то время как сложность матричного алгоритма имеет квадратичный рост, что наглядно изображено на рисунке 4.5.

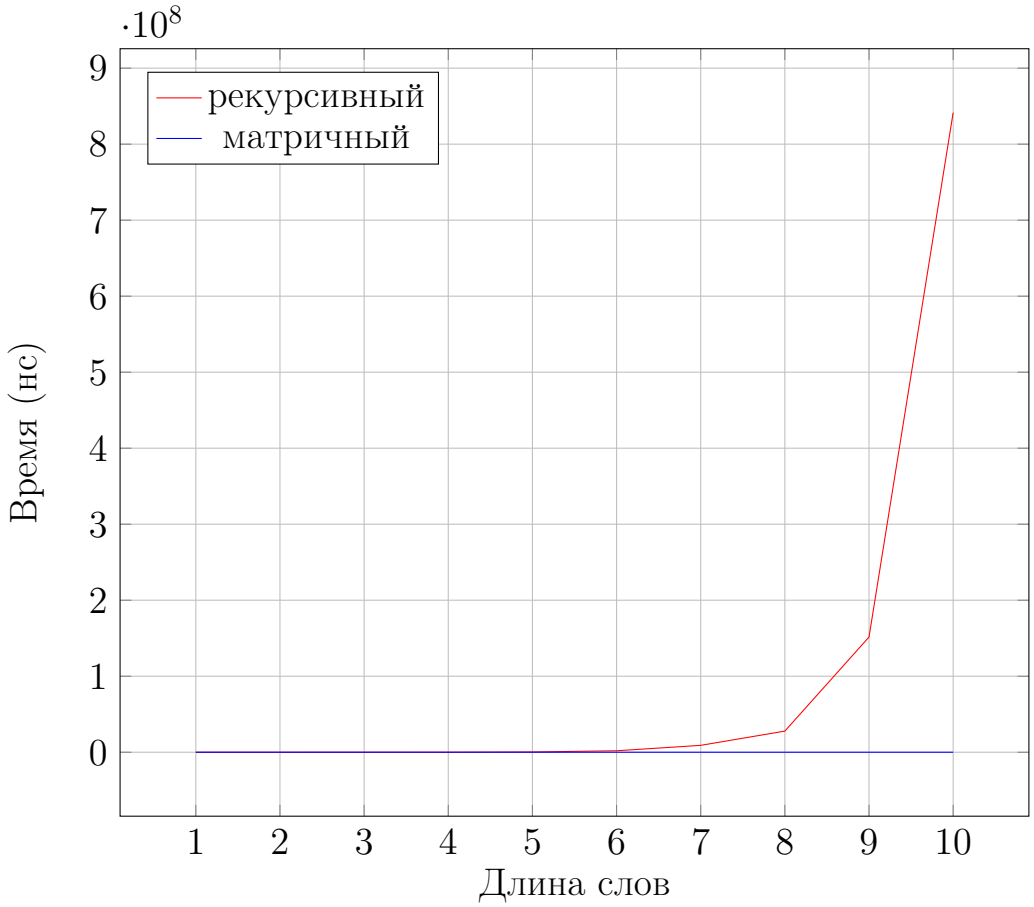

Рисунок 4.6 — График времени работы реализаций рекурсивного и матричного алгоритмов Дамерау-Левенштейна

4.3 Вывод

Как итог, была подтверждена корректная работоспособность реализованной программы нахождения расстояний Левенштейна и Дамерау-Левенштейна и доказаны тезисы, составленные в результате анализа этих алгоритмов.

ЗАКЛЮЧЕНИЕ

В ходе выполнения данной лабораторной работы мной был изучен метод динамического программирования на материале алгоритмов поиска редакционного расстояния. Кроме того, были изучены непосредственно алгоритмы поиска редакционного расстояния, проведён их анализ и сравнение, успешно реализована и протестирована программа, осуществляющая этот поиск, проведены эксперименты, в ходе которых были подтверждены полученные в ходе анализа тезисы.

При сравнении данных алгоритмов пришли к следующим выводам:

а) Рекурсивный алгоритм является самым медленным, гораздо быстрее использовать алгоритмы матричные.

б) Дамерау-Левенштейна проигрывает обычному Левенштейну на 20% на длинах слов, которые больше 200, но цена ошибки, в некоторых случаях, у него меньше.