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 Котляров Н.А.

Группа ИУ7-51Б

Оценка (баллы)

Преподаватель Волкова Л.Л.

Москва — 2022 г.

Оглавление

Введение **3**

1 Аналитическая часть **5**

1.1 Рекурсивный алгоритм нахождения расстояния Дамерау — Левенштейна 6

1.2 Матричный алгоритм 6

1.3 Рекурсивный алгоритм с использованием кеша 7

1.4 Расстояние Левенштейна 7

2 Конструкторская часть **10**

2.1 Требования к вводу 10

2.2 Требования к программе 10

2.3 Алгоритмы нахождения расстояния Дамерау – Левенштейна 10

2.4 Алгоритм нахождения расстояния Левенштейна 13

2.5 Вывод 15

3 Технологическая часть **16**

3.1 Требования к ПО 16

3.2 Средства реализации 16

3.3 Сведения о модулях программы 16

3.4 Листинг кода 17

3.5 Функциональные тесты 20

4 Исследовательская часть **22**

4.1 Технические характеристики 22

4.2 Демонстрация работы программы 22

4.3 Время выполнения алгоритмов 23

4.4 Использование памяти 26

Заключение **29**

Литература **30**

Введение

Целью данной лабораторной работы является разработка, реализация и исследование алгоритмов нахождения расстояний Левенштейна и Дамерау–Левенштейна.

Расстояние Левенштейна (редакционное расстояние, дистанция редактирования) — метрика, измеряющая разность между двумя последовательностями символов. Она определяется как минимальное количество односимвольных операций (вставки, удаления, замены), необходимых для превращения одной строки в другую. В общем случае, операциям, используемым в этом преобразовании, можно назначить разные цены. Это расстояние широко используется в теории информации и компьютерной лингвистике.

Расстояние Левенштейна [1] и его обобщения активно применяются в решении следующих задач:

- исправления ошибок в слове (в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи);
- сравнения текстовых файлов утилитой `diff` и ей подобными (здесь роль «символов» играют строки, а роль «строк» — файлы);
- сравнения геномов, хромосом и белков в биоинформатике.

Расстояние Дамерау — Левенштейна (названо в честь учёных Фредерика Дамерау и Владимира Левенштейна) — это мера разницы двух строк символов, определяемая как минимальное количество операций вставки, удаления, замены и транспозиции (перестановки двух соседних символов), необходимых для перевода одной строки в другую. Является модификацией расстояния Левенштейна, так как к операциям вставки, удаления и замены символов, определённых в расстоянии Левенштейна добавлена операция транспозиции (перестановки) символов.

Задачи данной лабораторной следующие:

1) изучение расстояний Левенштейна и Дамерау-Левенштейна;

2) применение метода динамического программирования для реализации алгоритмов;

3) получение практических навыков реализации алгоритмов Левенштейна и Дамерау — Левенштейна;

4) сравнительный анализ алгоритмов определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);

5) экспериментальное подтверждение различий во временно́й эффективности алгоритмов определения расстояния между строками при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк;

6) описание и обоснование полученных результатов в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

1 Аналитическая часть

В этом разделе будут представлены описания алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна и их практическое применение.

Расстояние Дамерау-Левенштейна [2] - это мера разницы двух строк символов, определяемая как минимальное количество операций вставки, удаления, замены и транспозиции (перестановки двух соседних символов), необходимых для перевода одной строки в другую. Является модификацией расстояния Левенштейна: к операциям вставки, удаления и замены символов, определённых в расстоянии Левенштейна добавлена операция транспозиции (перестановки) символов.

Каждая операция имеет свою цену (штраф). Редакционным предписанием называется последовательность действий, необходимых для получения из первой строки вторую, и минимизирующую суммарную цену.

Введем следующие обозначения:

- D (англ. delete) — удаление ($w(a, \lambda) = 1$);
- I (англ. insert) — вставка ($w(\lambda, b) = 1$);
- R (англ. replace) — замена ($w(a, b) = 1,\ a \neq b$);
- T (англ. transposition) — транспозиция ($w(ab, ba) = 1,\ a \neq b$);
- M (англ. match) - совпадение ($w(a, a) = 0$).

1.1 Рекурсивный алгоритм нахождения расстояния Дамерау — Левенштейна

Расстояние Дамерау — Левенштейна может быть найдено по формуле

$$d_{a,b}(i,j) = \begin{cases} \max(i,j), \text{если } \min(i,j) = 0, \\ \min\{ \\ \quad d_{a,b}(i,j-1) + 1, \\ \quad d_{a,b}(i-1,j) + 1, \\ \quad d_{a,b}(i-1,j-1) + m(a[i],b[j]), \\ \quad \begin{cases} d_{a,b}(i-2,j-2) + 1, & \text{если } i,j > 1; \\ & a[i] = b[j-1]; \\ & b[j] = a[i-1] \\ \infty, & \text{иначе} \end{cases} \\ \}, \text{иначе} \end{cases} \tag{1.1}$$

Функция m определена как:

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

1.2 Матричный алгоритм

При больших i, j прямая реализация формулы 1.4 может быть малоэффективна по времени исполнения, так как множество промежуточных значений $D(i,j)$ вычисляются не по одному разу. Для оптимизации нахождения можно использовать матрицу для хранения соответствующих промежуточных значений.

Необходима матрица размером $(length(S1)+1)$x$((length(S2)+1)$, где $length(S)$ — длина строки S. Значение в ячейке $[i,j]$ равно значению $D(S1[1...i], S2[1...j])$. Первая строка и первый столбец тривиальны.

Всю матрицу (за исключением первого столбца и первой строки) требуется заполнить в соответствии с формулой 1.3.

$$A[i][j] = min \begin{cases} A[i-1][j] + 1, \\ A[i][j-1] + 1, \\ A[i-1][j-1] + m(S1[i], S2[j]), \\ \begin{cases} A[i-2][j-2] + 1, & \text{если } i,j > 1; \\ & S1[i] = S2[j-1]; \\ & S2[j] = S1[i-1] \\ \infty, & \text{иначе} \end{cases} \end{cases} \tag{1.3}$$

В результате расстоянием Дамерау — Левенштейна будет ячейка матрицы с индексами $i = length(S1)$ и $j = length(S2)$.

1.3 Рекурсивный алгоритм с использованием кеша

Рекурсивный алгоритм заполнения можно оптимизировать по трудоемкости с использованием кеша [3]. В качестве кеша используется матрица. Суть данной оптимизации заключается в параллельном 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, для них расстояние не находится и алгоритм переходит к следующему шагу.

1.4 Расстояние Левенштейна

Расстояние Левенштейна [1] между двумя строками — это минимальное количество операций вставки, удаления и замены, необходимых для превращения одной строки в другую.

Каждая операция имеет свою цену (штраф). Редакционным предписанием называется последовательность действий, необходимых для получения из первой строки вторую, и минимизирующую суммарную цену. Суммарная цена есть искомое расстояние Левенштейна.

Введем следующие обозначений:

- D (англ. delete) — удаление ($w(a, \lambda) = 1$);
- I (англ. insert) — вставка ($w(\lambda, b) = 1$);
- R (англ. replace) — замена ($w(a, b) = 1,\ a \neq b$);
- M (англ. match) - совпадение ($w(a, a) = 0$).

Пусть S_1 и S_2 — две строки (длиной M и N соответственно) над некоторым алфавитом, тогда расстояние Левенштейна можно подсчитать по следующей рекуррентной формуле:

$$D(i,j) = \begin{cases} 0 & \text{i} = 0,\ \text{j} = 0 \\ i & \text{j} = 0,\ \text{i} > 0 \\ j & \text{i} = 0,\ \text{j} > 0 \\ \min\{ & \\ \quad D(i, j-1) + 1 & \\ \quad D(i-1, j) + 1 & \text{i} > 0,\ \text{j} > 0 \\ \quad D(i-1, j-1) + m(a[i], b[j]) & \\ \} & \end{cases} \tag{1.4}$$

Рекурсивный алгоритм реализует формулу 1.4. Функция D должна быть составлена составлена таким образом, что для перевода из строки a в строку b требуется выполнить последовательно некоторое количество операций (удаление, вставка, замена) в некоторой последовательности. Полагая, что a', b' — строки a и b без последнего символа соответственно, цена преобразования из строки a в строку b может быть получено как одно из следующих значений :

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

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

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

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

Минимальной ценой преобразования будет минимальное значение приведенных вариантов.

Вывод

Формулы Левенштейна и Дамерау — Левенштейна для рассчета расстояния между строками задаются рекуррентно, а следовательно, алгоритмы могут быть реализованы рекурсивно или итерационно.

2 Конструкторская часть

В этом разделе будут приведены требования к вводу и программе, а также схемы алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна.

2.1 Требования к вводу

К вводу программы должны быть предъявлены следующие требования.

1. На вход подаются две строки.
2. Буквы верхнего и нижнего регистров считаются различными.

2.2 Требования к программе

К программе должны быть предъявлены следующие требования.

1. Две пустые строки - корректный ввод, программа не должна аварийно завершаться.
2. На выход программа должна вывести число - расстояние Левенштейна (Дамерау-Левенштейна), матрицу при необходимости.

2.3 Алгоритмы нахождения расстояния Дамерау – Левенштейна

На рисунке 2.1 приведена схема рекурсивного алгоритма нахождения расстояния Дамерау – Левенштейна.

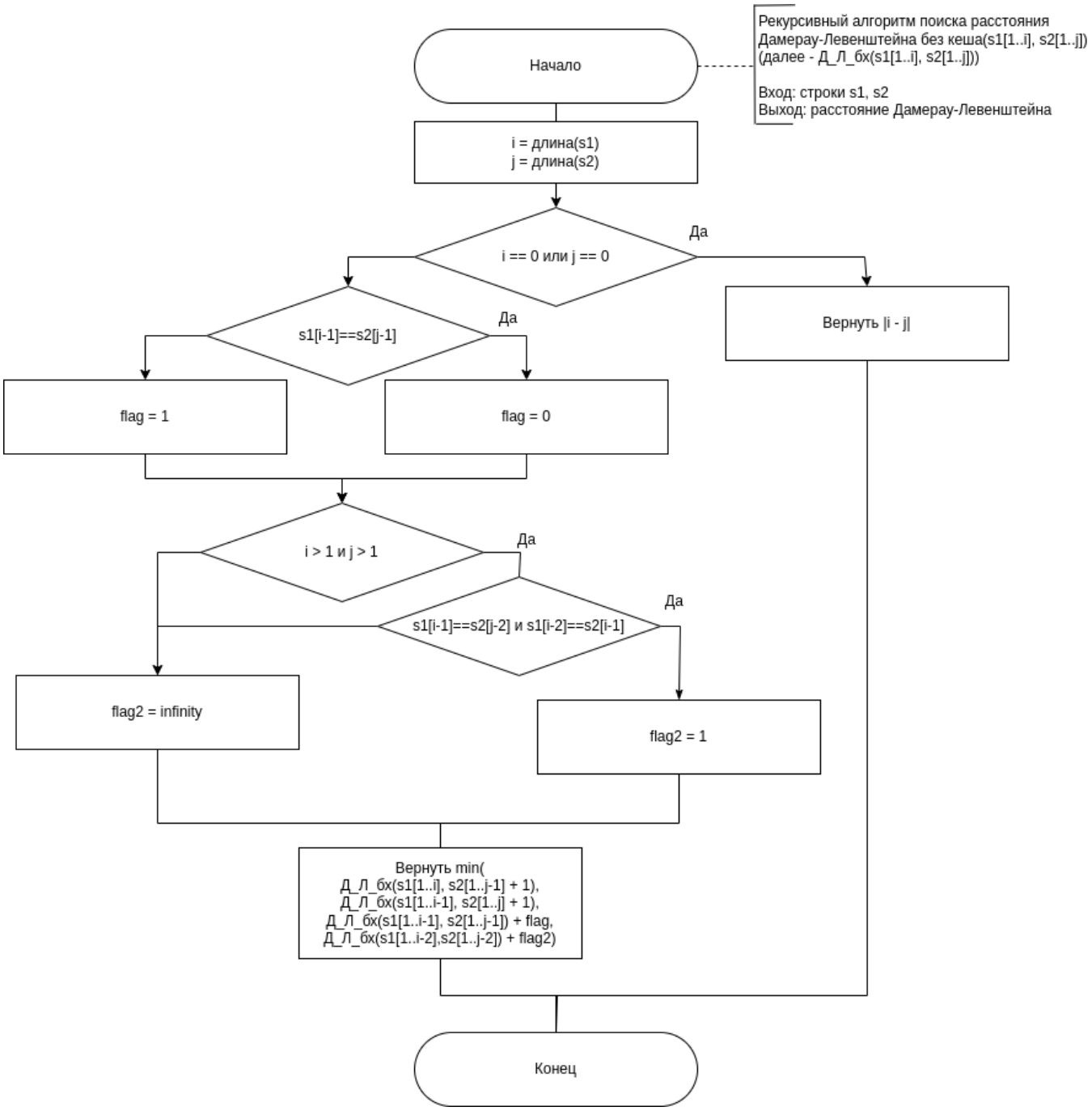

Рисунок 2.1 – Схема рекурсивного алгоритма нахождения расстояния Дамерау – Левенштейна

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

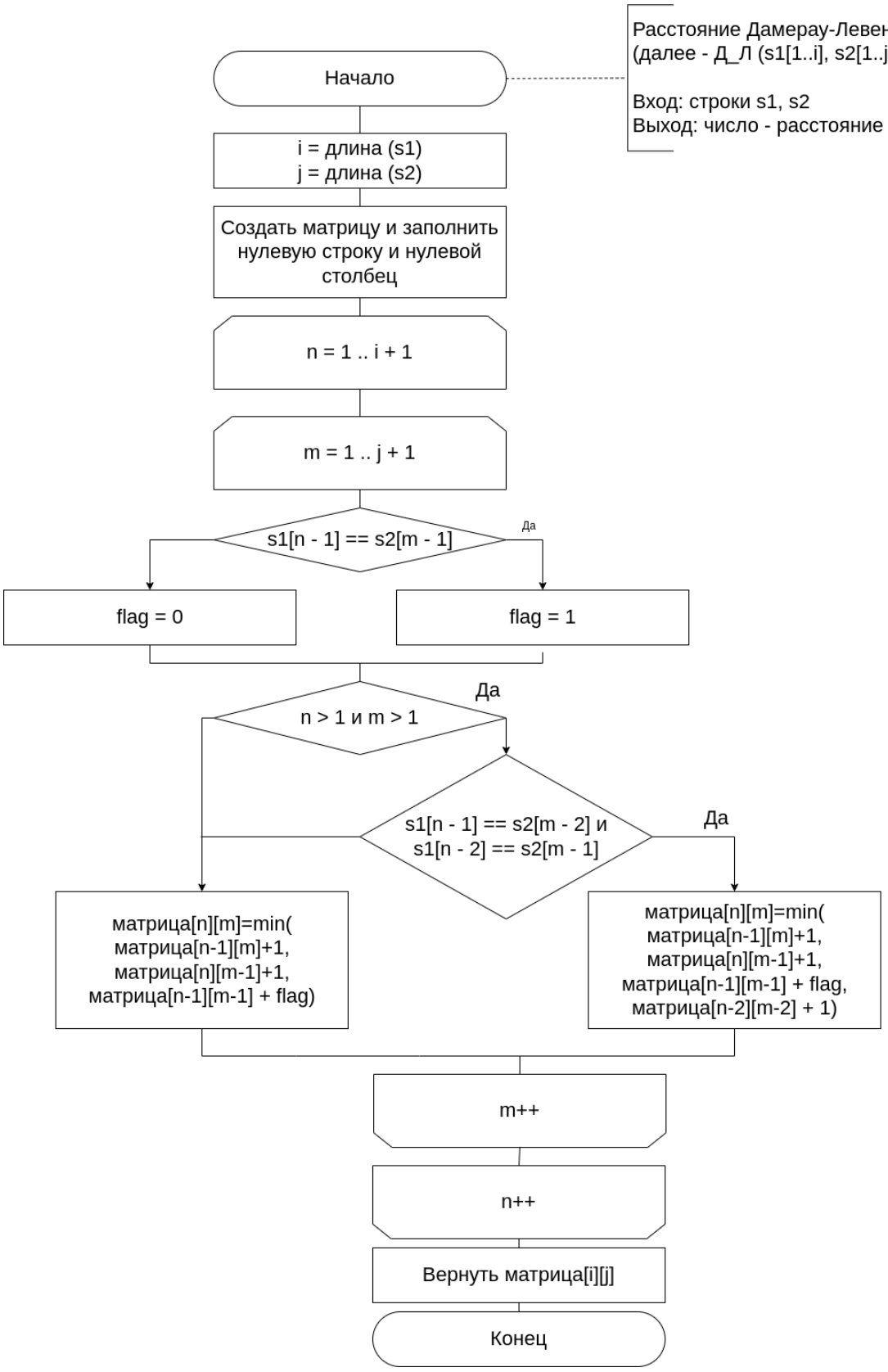

Рисунок 2.2 – Схема алгоритма нахождения расстояния Дамерау – Левенштейна с заполнением матрицы

На рисунке 2.3 приведена схема рекурсивного алгоритма нахождения расстояния Дамерау – Левенштейна с использованием кеша в виде матри-

цы.

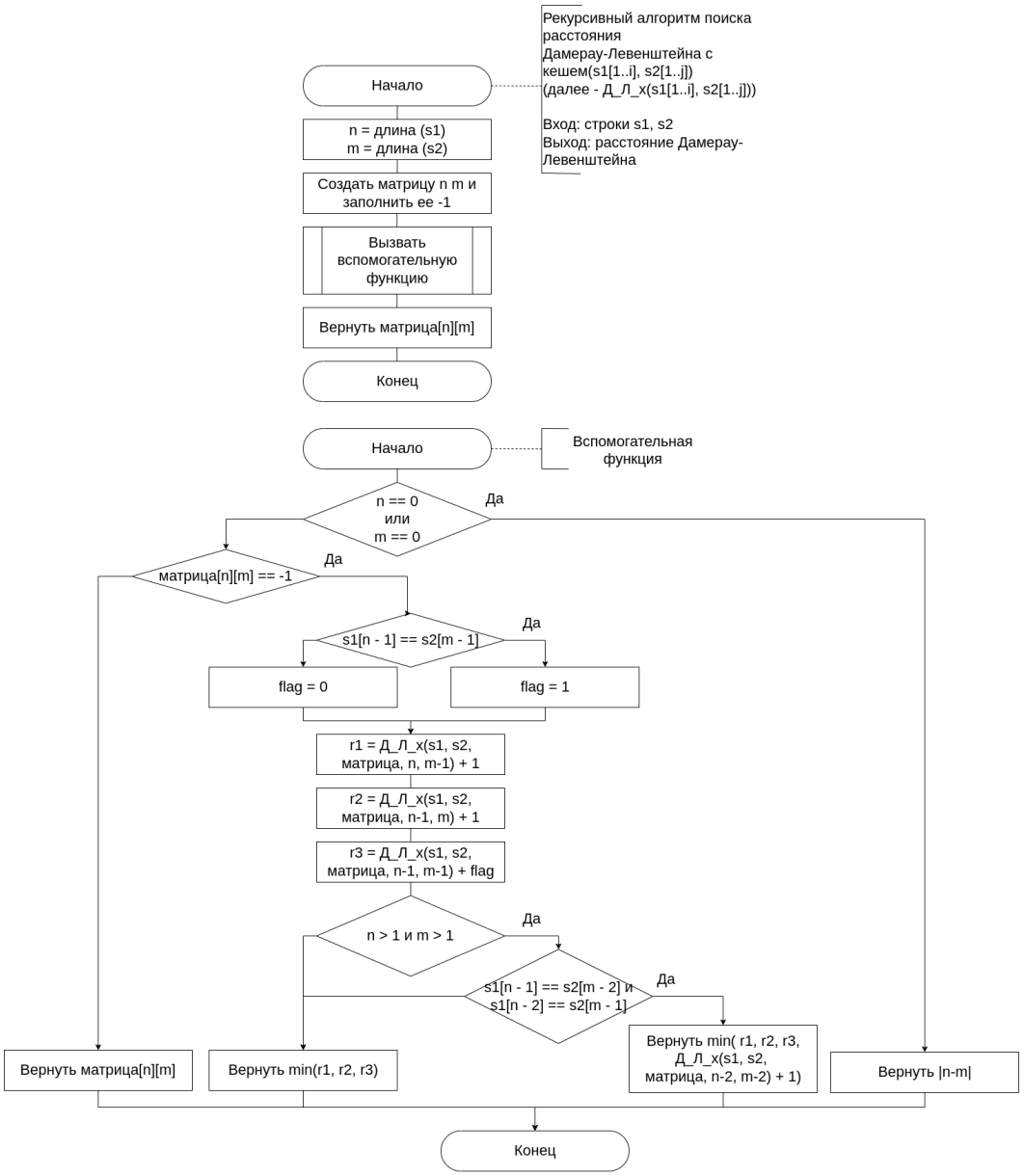

Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Дамерау – Левенштейна с использованием кеша в виде матрицы

2.4 Алгоритм нахождения расстояния Левенштейна

На рисунке 2.4 приведена схема рекурсивного алгоритма нахождения расстояния Левенштейна.

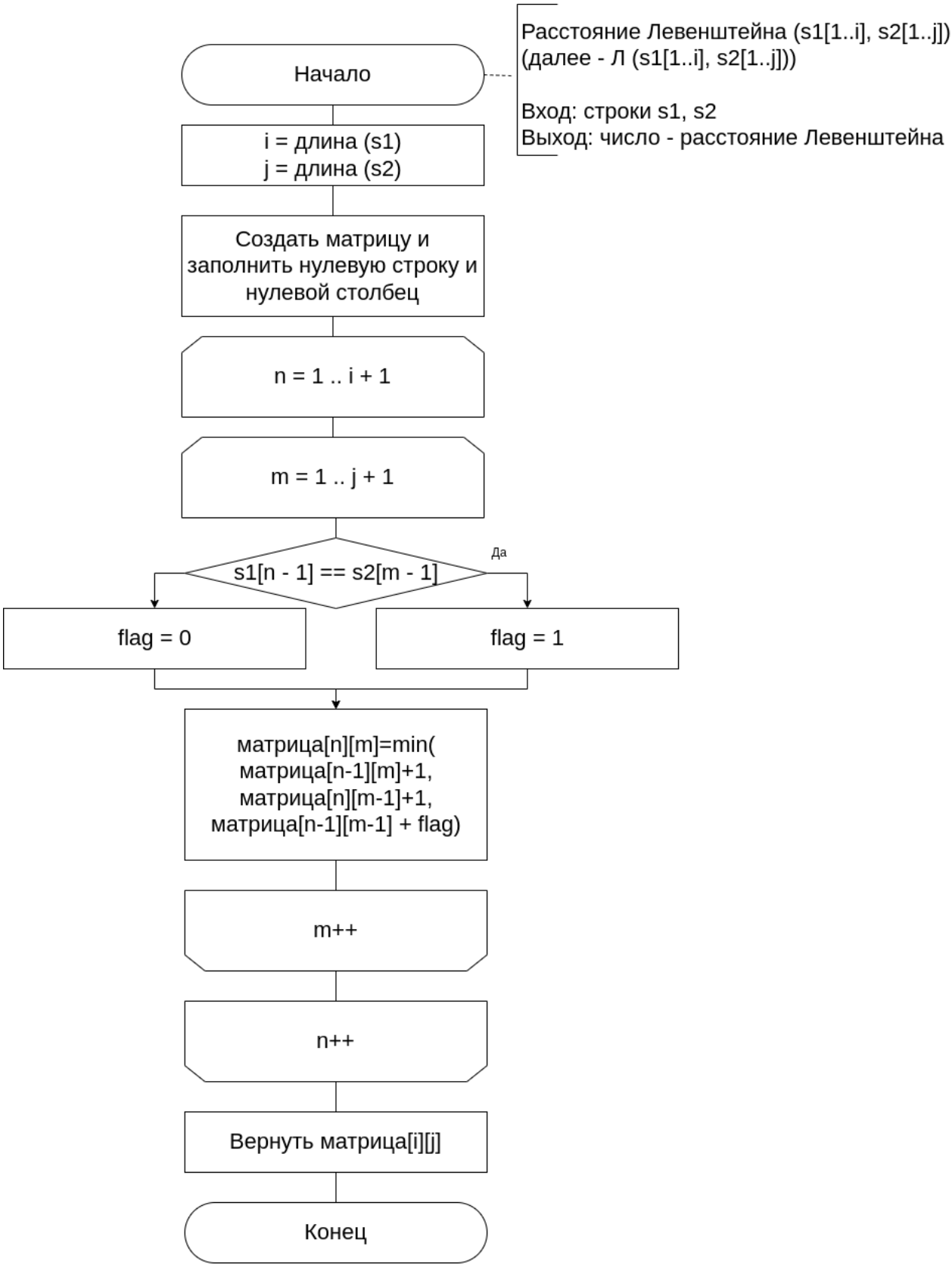

Рисунок 2.4 – Схема алгоритма нахождения расстояния Левенштейна с заполнением матрицы

2.5 Вывод

Перечислены требования к вводу и программе, а также на основе теоретических данных, полученных из аналитического раздела, были построены схемы требуемых алгоритмов.

3 Технологическая часть

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

3.1 Требования к ПО

К программе предъявляется ряд требований:

- реализованны все алгоритмы, соответствующие варианту;
- на вход подаются две строки на русском или английском языке в любом регистре;
- программа выдает искомое расстояние для выбранного метода (выбранных методов) и матрицы расстояний для матричных реализаций.

3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язык программирования Golang [4].

В данном языке есть все требующиеся для данной лабораторной инструменты разработки.

Время работы алгоритмов будет измеряться с помощью команды time в Linux [5].

3.3 Сведения о модулях программы

Программа состоит из трех модулей:

1) main.go – главный файл программы;
2) algos.go – файл программы, в котором располагаются коды всех алгоритмов;

3) sub.go – файл программы, в котором располагаются вспомогательные функции.

3.4 Листинг кода

В листингах 3.1, 3.2, 3.3, 3.4 приведены реализации алгоритмов нахождения расстояния Левенштейна и Дамерау–Левенштейна.

Листинг 3.1 – Функция нахождения расстояния Дамерау–Левенштейна с использованием рекурсии

```
1  func dam_lev_recursive(str1, str2 string, output bool) int {
2      min_ret := 0
3      n := len(str1)
4      m := len(str2)
5      if n == 0 {
6          return m
7      }
8      if m == 0 {
9          return n
10     }
11     change := 0
12     if str1[n-1] != str2[m-1] {
13         change += 1
14     }
15     if n > 1 && m > 1 && str1[n-1] == str2[m-2] && str1[n-2] ==
           str2[m-1] {
16         min_ret = MinOf(dam_lev_recursive(str1[:n-1], str2,
               output)+1,
17         dam_lev_recursive(str1, str2[:m-1], output)+1,
18         dam_lev_recursive(str1[:n-1], str2[:m-1], output)+change,
19         dam_lev_recursive(str1[:n-2], str2[:m-2], output)+1)
20     } else {
21         min_ret = MinOf(dam_lev_recursive(str1[:n-1], str2,
               output)+1,
22         dam_lev_recursive(str1, str2[:m-1], output)+1,
23         dam_lev_recursive(str1[:n-1], str2[:m-1], output)+change)
24     }
25     return min_ret
26 }
```

Листинг 3.2 – Функция нахождения расстояния Дамерау–Левенштейна с использованием матрицы.

```
1  func dam_lev_matrix(str1, str2 string, output bool) int {
2      min_ret := 0
3      n := len(str1)
4      m := len(str2)
5      matr := createMatr(n+1, m+1)
6      for i := 1; i < n+1; i++ {
7          for j := 1; j < m+1; j++ {
8              add, delete, change := matr[i-1][j]+1,
                   matr[i][j-1]+1, matr[i-1][j-1]
9
10             if str1[i-1] != str2[j-1] {
11                 change += 1
12             }
13
14             matr[i][j] = MinOf(add, delete, change)
15
16             if i > 1 && j > 1 && str1[i-1] == str2[j-2] &&
                   str1[i-2] == str2[j-1] {
17                 matr[i][j] = MinOf(matr[i][j], matr[i-2][j-2]+1)
18             }
19         }
20     }
21     return min_ret
22 }
```

Листинг 3.3 – Функция нахождения расстояния Дамерау–Левенштейна с использованием рекурсии с кешем.

```
1  func dam_lev_rec_cash_help(str1, str2 string, matr MatrInt) {
2      len1 := len(str1)
3      len2 := len(str2)
4
5      if len1 == 0 {
6          matr[len1][len2] = len2
7      } else if len2 == 0 {
8          matr[len1][len2] = len1
9      } else {
10         //insert
11         if matr[len1][len2-1] == INF {
12             dam_lev_rec_cash_help(str1, str2[:len2-1], matr)
```

```
13          }
14          //delete
15          if matr[len1-1][len2] == INF {
16              dam_lev_rec_cash_help(str1[:len1-1], str2, matr)
17          }
18          //replase
19          if matr[len1-1][len2-1] == INF {
20              dam_lev_rec_cash_help(str1[:len1-1], str2[:len2-1],
                    matr)
21          }
22
23          change := 0
24          if str1[len1-1] != str2[len2-1] {
25              change += 1
26          }
27
28          matr[len1][len2] = MinOf(matr[len1][len2-1]+1,
29          matr[len1-1][len2]+1,
30          matr[len1-1][len2-1]+change)
31
32          if len1 > 1 && len2 > 1 && str1[len1-1] == str2[len2-2]
              && str1[len1-2] == str2[len2-1] {
33              matr[len1][len2] = MinOf(matr[len1][len2],
                    matr[len1-2][len2-2]+1)
34          }
35      }
36  }
37
38  func dam_lev_rec_cash(str1, str2 string, output bool) int {
39      n := len(str1)
40      m := len(str2)
41
42      matr := createInfMatrix(n+1, m+1)
43      dam_lev_rec_cash_help(str1, str2, matr)
44
45      if output {
46          printMatr(matr, n+1, m+1)
47      }
48
49      return matr[n][m]
50  }
```

Листинг 3.4 – Функция нахождения расстояния Левенштейна с использованием итераций.

```
1  func lev_matrix(str1, str2 string, output bool) int {
2      n := len(str1)
3      m := len(str2)
4
5      matr := createMatr(n+1, m+1)
6      for i := 1; i < n+1; i++ {
7          for j := 1; j < m+1; j++ {
8              add, delete, change := matr[i-1][j]+1,
                   matr[i][j-1]+1, matr[i-1][j-1]
9
10             if str1[i-1] != str2[j-1] {
11                 change += 1
12             }
13
14             matr[i][j] = MinOf(add, delete, change)
15         }
16     }
17
18     if output {
19         printMatr(matr, n+1, m+1)
20     }
21
22     return matr[n][m]
23 }
```

3.5 Функциональные тесты

В таблице 3.1 приведены функциональные тесты для алгоритмов вычисления расстояния Левенштейна (в таблице столбец подписан "Левенштейн") и Дамерау — Левенштейна (в таблице - "Дамерау-Л."). Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты

	Входные данные		Ожидаемый результат	
№	Строка 1	Строка 2	Левенштейн	Дамерау-Л.
1	скат	кот	2	2
2	машина	малина	1	1
3	дворик	доврик	2	1
4	"пустая строка"	университет	11	11
5	сентябрь	"пустая строка"	8	8
8	тело	телодвижение	8	8
9	ноутбук	планшет	7	7
10	глина	малина	2	2
11	рекурсия	ркерусия	3	2
12	браузер	баурзер	2	2
13	bring	brought	4	4
14	moment	minute	4	4
15	person	eye	5	5
16	week	weekend	3	3
17	city	town	4	4

Вывод

Были разработаны и протестированы алгоритмы: нахождения расстояния Дамерау – Левенштейна рекурсивно, с заполнением матрицы и рекурсивно с заполнением матрицы, а также нахождения расстояния Левенштейна с заполнением матрицы.

4 Исследовательская часть

В данном разделе будут приведены примеры работы программ, постановка эксперимента и сравнительный анализ алгоритмов на основе полученных данных.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- операционная система: Manjaro xfce [6] Linux [7] x86_64;
- память: 8 Гб;
- мобильный процессор AMD Ryzen™ 7 3700U @ 2.3Гц [8].

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

4.2 Демонстрация работы программы

Программа получает на вход 2 слова и выдает 4 расстояния, соответствующие расстояниям Дамерау – Левенштейна, полученным матричным, рекурсивным и рекурсивным с кешем алгоритмами и расстоянию Левенштейна, полученному матричным алгоритмом.

На рисунке 4.1 представлен результат работы программы.

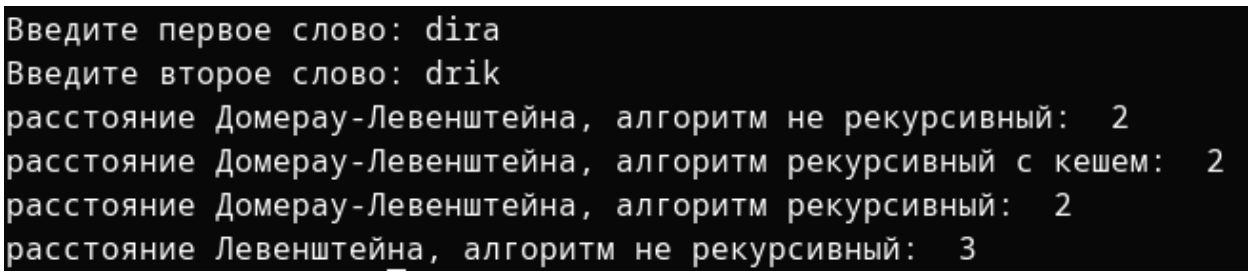

Рисунок 4.1 – Пример работы программы

4.3 Время выполнения алгоритмов

Алгоритмы тестировались при помощи команды Linux time. Данная команда всегда возвращает значения времени, а именно реальное время, системное и пользовательское процессорное временя текущего процессора, типа float в секундах.

Замеры времени для каждой длины слов проводились 1000000 раз, не считая рекурсивного алгоритма: для длины 6 – 100000, длины 7 и 8 – 10000, длины 9 – 1000. В качестве результата взято среднее время работы алгоритма на данной длине слова.

Оба слова имеют одинаковую длину в символах.

Результаты замеров приведены в таблице 4.1 (время в мкс).

Таблица 4.1 – Результаты замеров времени

Длина	Д.-Л.(матр.)	Д.-Л.(рек с матр.)	Д.-Л.(рек)	Л.(матр.)
0	0.585	0.726	0.199	0.575
1	0.770	0.876	0.235	0.768
2	0.879	1.126	0.404	0.873
3	1.328	1.407	0.596	1.218
4	1.385	1.763	1.766	1.273
5	1.617	2.200	7.747	1.602
6	1.782	2.605	43.69	1.697
7	2.476	3.273	263.9	2.449
8	2.845	3.892	1281.0	2.651
9	3.379	4.573	7181.0	3.155

На рисунке 4.2 представлены результаты сравнения рекурсивных реализаций алгоритмов поиска расстояния Дамерау – Левенштейна с исполь-

зованием кеша и без. На графике видно, что полученные результаты частично накладываются друг на друга (до длины равной 4).

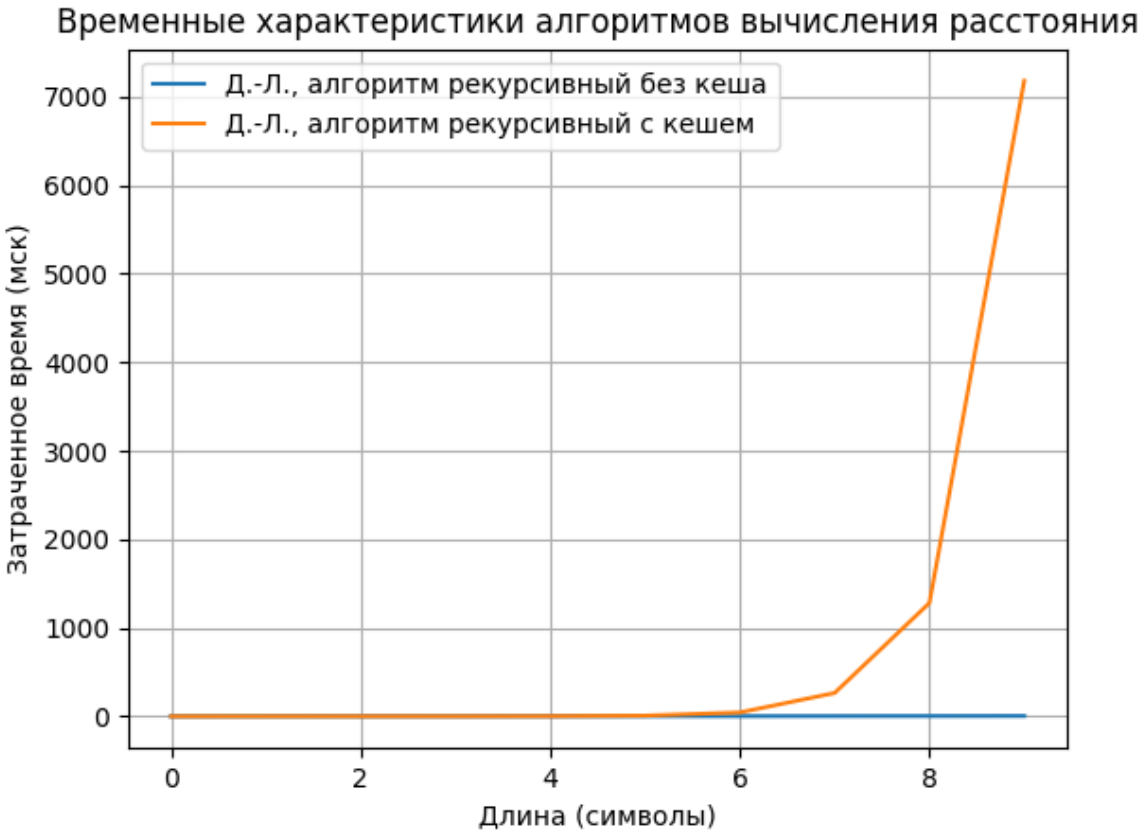

Рисунок 4.2 – Сравнения рекурсивных алгоритмов поиска расстояния Левенштейна с использованием кеша и без

На рисунке 4.3 представлены результаты сравнения итерационных реализаций алгоритмов поиска расстояния Левенштейна и Дамерау – Левенштейна. На графике видно, что полученные результаты частично накладываются друг на друга (до длины равной 7).

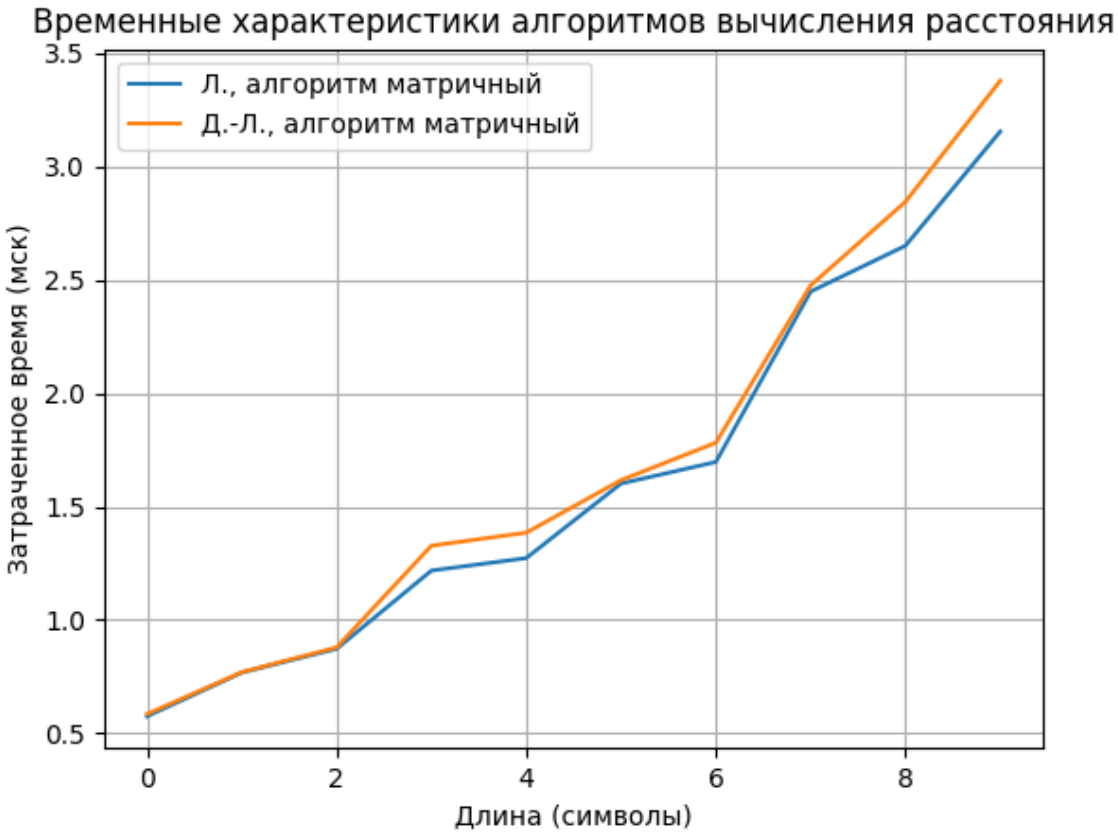

Рисунок 4.3 – Сравнения итерационных алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна

На рисунке 4.4 представлены результаты сравнения матричной реализаций алгоритма поиска расстояния Дамерау – Левенштейна и рекурсивного алгоритма поиска расстояния Дамерау – Левенштейна с использованием кеша.

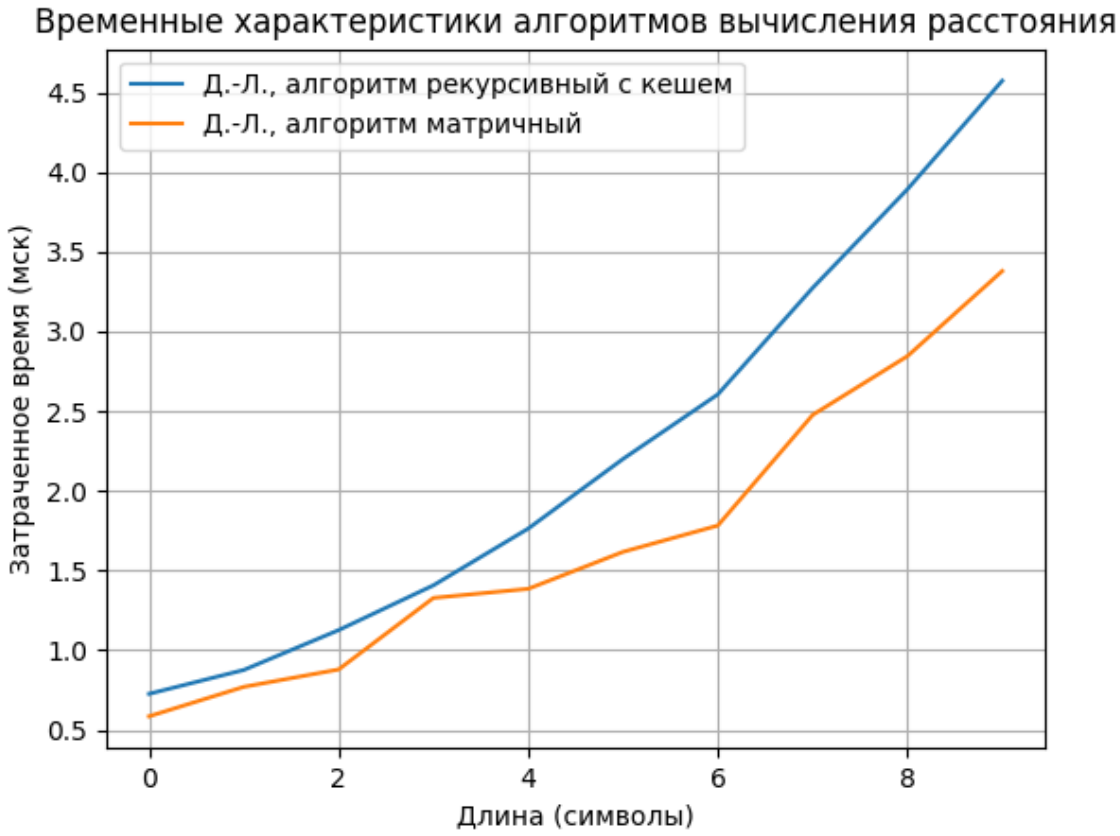

Рисунок 4.4 – Сравнения матричного алгоритма поиска расстояния Дамерау – Левенштейна и рекурсивного алгоритма поиска расстояния Дамерау – Левенштейна с использованием кеша

4.4 Использование памяти

Алгоритмы нахождения расстояний Левенштейна и Дамерау – Левенштейна не отличаются друг от друга с точки зрения использования памяти, поэтому достаточно рассмотреть лишь разницу рекурсивной и матричной реализаций данных алгоритмов.

Пусть длина строки S1 - n, длина строки S2 - m, тогда затраты памяти на приведенные выше алгоритмы будут следующими.

- Матричный алгоритм поиска расстояния Левенштейна:
  - строки S1, S2 - (m + n) * sizeof(char);
  - матрица - ((m + 1) * (n + 1)) * sizeof(int);
  - текущая строка матрицы - (n + 1) * sizeof(int);

- длины строк - 2 * sizeof(int);
- вспомогательные переменные - 3 * sizeof(int).

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

- Рекурсивный алгоритм поиска расстояния Дамерау – Левенштейна (для каждого вызова):
  - строки S1, S2 – (m + n) * sizeof(char);
  - длины строк – 2 * sizeof(int);
  - вспомогательные переменные – 2 * sizeof(int);
  - размер аргументов функции – 2 * 24;
  - размер адреса возврата – 4.

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк. Таким образом, общая затраченная память в рекурсивном алгоритме будет равна $m+n+(2*4+2*4+4+2*24)*(n+m) = 85m + 85n$.

- Рекурсивный алгоритм поиска расстояния Дамерау – Левенштейна с использованием кеша (для каждого вызова): Для всех вызовов еще память для хранения самой матрицы - ((m + 1) * (n + 1)) * sizeof(int)
  - строки S1, S2 - (m + n) * sizeof(char);
  - длины строк - 2 * sizeof(int);
  - вспомогательные переменные - 1 * sizeof(int);
  - размер аргументов функции – 2 * 24;
  - ссылка на матрицу - 8 байт;
  - размер адреса возврата – 4.

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк. Таким образом, общая затраченная память в рекурсивном алгоритме будет равна $m + n + 4 * (m + 1) * (n + 1) + (2 * 4 + 4 + 4 + 2 * 24) * (n + m) = 4mn + 69m + 69n + 4$.

- Матричный алгоритм поиска расстояния Дамерау – Левенштейна:
  - строки S1, S2 - (m + n) * sizeof(char);
  - матрица - ((m + 1) * (n + 1)) * sizeof(int);
  - длины строк - 2 * sizeof(int);
  - вспомогательные переменные - 3 * sizeof(int).

Таким образом, общая затраченная память в рекурсивном алгоритме будет равна $m + n + 4 * (m + 1) * (n + 1) + 6 * 4 = 4mn + 5m + 5n + 28$.

Вывод

Рекурсивный алгоритм нахождения расстояния Дамерау – Левенштейна работает на порядок дольше итеративных реализаций, время его работы увеличивается в геометрической прогрессии. На словах длиной 9 символов, матричная реализация алгоритма нахождения расстояния Дамерау – Левенштейна превосходит по времени работы рекурсивную на несколько порядков.

Рекурсивный алгоритм с заполнением матрицы превосходит простой рекурсивный и сравним по времени работы с матричными алгоритмами.

Алгоритм нахождения расстояния Левенштейна по времени выполнения сопоставим с алгоритмом нахождения расстояния Дамерау – Левенштейна. В нём отсутствует дополнительная проверка, позволяющая находить ошибки пользователя, связанные с неверным порядком букв, в связи с чем он работает незначительно быстрее, чем алгоритм нахождения расстояния Дамерау – Левенштейна.

Но по расходу памяти (при очень больших длинах строк) матричные алгоритмы проигрывают рекурсивному: максимальный размер используемой памяти в них растёт как произведение длин строк, в то время как у рекурсивного алгоритма — как сумма длин строк.

Заключение

В ходе выполнения лабораторной работы была достигнута цель работа: были разработаны алгоритмы нахождения расстояний Левенштейна и Дамерау–Левенштейна.

Все поставленные задачи решены:

- реализовать алгоритмы нахождения расстояний Левенштейна и Дамерау – Левенштейна;
- реализованы алгоритмы поиска расстояния Дамерау – Левенштейна с заполнением матрицы, с использованием рекурсии и с помощью рекурсивного заполнения матрицы (рекурсивный с использованием кеша);
- реализован алгоритм поиска расстояния Левенштейна с использованием матрицы;
- проведен сравнительный анализ линейной и рекурсивной реализаций алгоритмов определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);
- проведено экспериментальное подтверждение различий во временной эффективности рекурсивной и нерекурсивной реализаций алгоритмов при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализации на различных длинах строк;
- подготовлен отчет о лабораторной работе.

Экспериментально было подтверждено различие во временной эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения между строками при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализаций на различных длин строк.

По итогу исследований было выявлено, что матричная реализация алгоритмов нахождения расстояний заметно выигрывает по времени при росте строк, но проигрывает по количеству затрачиваемой памяти.

Литература

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

[2] Черненький В. М. Гапанюк Ю. Е. Методика идентификации пассажира по установочным данным / Вестник МГТУ им. Н.Э. Баумана. Сер. “Приборостроение”. — М.: Издательство МГТУ им. Н.Э. Баумана, 2012. Т. 163. С. 30–34.

[3] Кеш – Википедия [Электронный ресурс]. Режим доступа: `https://ru.wikipedia.org/wiki/\T2A\CYRK\T2A\cyrerev\T2A\cyrsh` (дата обращения: 03.10.2022).

[4] The Go Programming Language Documentation [Электронный ресурс]. Режим доступа: `https://go.dev/doc/` (дата обращения: 05.10.2022).

[5] time — Linux Time Command | Linuxize [Электронный ресурс]. Режим доступа: `https://linuxize.com/post/linux-time-command/` (дата обращения: 05.10.2022).

[6] Manjaro [Электронный ресурс]. Режим доступа: `https://manjaro.org/` (дата обращения: 03.10.2022).

[7] Linux – Википедия [Электронный ресурс]. Режим доступа: `https://ru.wikipedia.org/wiki/Linux` (дата обращения: 04.10.2022).

[8] Мобильный процессор AMD Ryzen™ 7 3700U [Электронный ресурс]. Режим доступа: `https://www.amd.com/ru/products/apu/amd-ryzen-7-3700u` (дата обращения: 04.10.2022).