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»
ДИСЦИПЛИНА «Анализ алгоритмов»

Лабораторная работа № 1

Дисциплина Анализ алгоритмов
Тема Расстояние Левенштейна и Дамерау-Левенштейна
Студент Боренко Анастасия
Группа ИУ7-52Б
Оценка (баллы) _____
Преподаватель Волкова Л.Л.

Москва — 2021г.

Содержание

Введение

Расстояние Левенштейна – это минимальное количество редакторских операций необходимое для превращения одной строки в другую. Редакторскими операциями считаются вставка, удаление, замена.

Расстояние Дамерау-Левенштейна включает также операцию перестановки 2 символов.

Расстояние Левенштейна применяется в следующих сферах:

- компьютерная лингвистика:
  - автозамена;
  - сравнение текстов;
  - поисковые системы;
- биоинформатика:
  - сравнение генов, хромосом и белков.

Целью данной лабораторной являются изучение метода динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна. Задачами данной лабораторной являются:

1. изучение алгоритмов Левейнштейна и Дамерау-Левенштейна нахождения расстояния между строками;
2. применение метода динамического программирования для матричных алгоритмов;
3. реализация указанных алгоритмов;

4. сравнительный анализ линейной и рекурсивной реализаций выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам(времени и памяти);

5. экспериментальное подтверждение различий во временной эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработанного времени выполнени7я реализации на варьирующихся длинах строк;

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

1. Аналитическая часть

Для поиска расстояний между строками необходимо определить расстояние между строками и разработать алгоритм, его нахождения.

1.1 Описание проблемы поиска расстояния Левенштейна

Расстояние Левенштейна – это минимальное количество редакторских операций необходимое для превращения одной строки в другую. [?] Редакторскими операциями считаются:

1. вставка(I - insert),
2. удаление (D - delete),
3. замена (R - replace).

Совпадение помечается буквой M - merge.

Найти расстояние Левенштейна можно тремя способами:

1. матрично;
2. рекурсивно;
3. рекурсивно с кэшем.

На рисунке 1.1 показан пример расчета расстояния Левенштейна.

Рис. 1.1: Пример расчета расстояния Левенштейна

1.2 Рекурсивный алгоритм поиска расстояния Левенштейна

Алгоритм можно описать следующей формулой $d(S_1, S_2) = D(M, N)$, где S_1 и S_2 - две строки длиной M и N над некоторым алфавитом.

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

где $m(a,b) = 0$, если $a = b$, иначе $m(a,b) = 1$, $min(a,b,c)$ - возвращает наименьший аргумент, $max(a,b,c)$ - возвращает набольший аргумент

1.3 Рекурсивный алгоритм поиска расстояния Левенштейна с кешем

В рекурсином алгоритме часто повторяются одинаковые действия, т.к. не храняться уже вычисленные значения. Проблема показана на рисунке 1.2. На рисунке красным выделены итерации, которые выполняются несколько раз, что требует дополнитель-

ные вычисления.

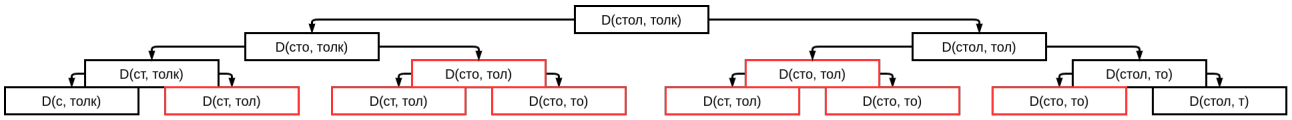

Рис. 1.2: Проблема рекурсивного метода

Для решения этой проблемы можно использовать кеширование. Необходимо запоминать строки, над которыми выполняется операция, и ее результат. Способ, использующий кеш, называется рекурсивный алгоритм поиска расстояния Левенштейна с кешем.

1.4 Матричный алгоритм поиска расстояния Левенштейна

На рисунке 1.3 показан пример расчета расстояния Левенштейна матричным способом.

		С	Т	О	Л
	0	1	2	3	4
Т	1	2	1	2	3
О	2	2	3	1	2
Л	3	3	3	2	1
К	4	4	4	3	2

Рис. 1.3: Пример расчета расстояния Левенштейна матричным способом

На рисунке 1.4 показан пример расчета расстояния Левенштейна матричным способом.

		С	Т	О	Л
	0	1	2	3	4
Т	1	2	1	2	3
О	2	2	3	1	2
Л	3	3	3	2	1
К	4	4	4	3	2

Рис. 1.4: Пример расчета расстояния Левенштейна матричным способом

1.5 Рекурсивный алгоритм поиска расстояния Дамерау - Левенштейна

Расстояние Дамерау-Левенштейна – это минимальное количество редакторских операций необходимое для превращения одной строки в другую. Редакторскими операциями считаются:

1. вставка(I - insert),
2. удаление (D - delete),
3. замена (R - replace),
4. перестановка 2 соседних символов (X - change).

Совпадение помечается буквой M - merge.

Расстояние Дамерау-Левенштейна в некоторых случаях может быть короче расстояния Левенштейна. (рис. 1.5).

Рис. 1.5: Сравнение расчета расстояний Левенштейна и Дамерау-Левенштейна

Алгоритм можно описать следующей формулой, где S_1 и S_2 - две строки длиной M и N над некоторым алфавитом. Тогда расстояние Дамеау-Левенштейна $d(S_1, S_2) = D(M, N)$.

$$
D = \begin{cases}
max(i,j), & \text{i = 0 or j = 0} \\
min\{ & \\
\quad D(i,j-1)+1, & \\
\quad D(i-1,j)+1, & \text{i > 1, j > 1 and } a_i = b_{j-1}, a_{i-1} = b_j \\
\quad D(i-1,j-1)+m(S_1[i],S_2[j]), & \\
\quad D(i-2,j-2)+1 & \\
\}, & \\
min\{ & \\
\quad D(i,j-1)+1, & \\
\quad D(i-1,j)+1, & \text{other} \\
\quad D(i-1,j-1)+m(S_1[i],S_2[j]) & \\
\}
\end{cases}
$$

где $m(a, b) = 0$, если $a = b$, иначе $m(a, b) = 1$, $min(a, b, c)$ - возвращает наименьший аргумент, $max(a, b, c)$ - возвращает набольший аргумент.

1.6 Вывод аналитической части

В данной работе стоит задача реализации следующих алгоритмов поиска расстояния Левенштейна: матричный алгоритм, рекурсивный алгоритм и рекурсивный алгоритм с кешем, и задача реализации рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна. Необходимо сравнить алгоритмы поиска расстояния Левенштейна по эффективности по времени и памяти.

Реализуем алгоритмы и сравним их затраченные на их выполнение время и память.

2. Конструкторская часть

2.1 Схемы алгоритмов

2.1.1 Схема матричного алгоритма поиска расстояния Левенштейна

На рисунке 2.1 показана схема алгоритма расчета расстояния Левенштейна матричным способом.

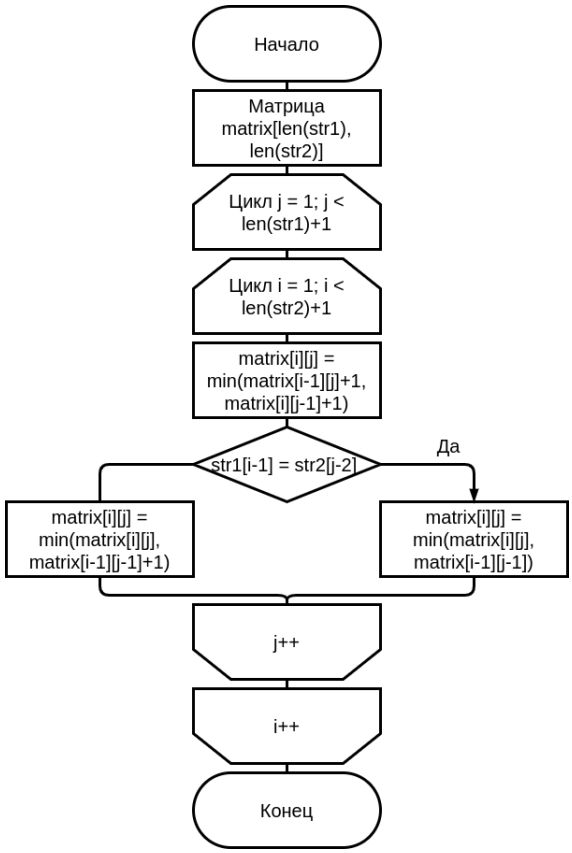

Рис. 2.1: Схема алгоритма нахождения расстояния Левенштейна матричным способом

2.1.2 Схема рекурсивного алгоритма поиска расстояния Левенштейна

На рисунке 2.2 показана схема алгоритма расчета расстояния Левенштейна с помощью рекурсии.

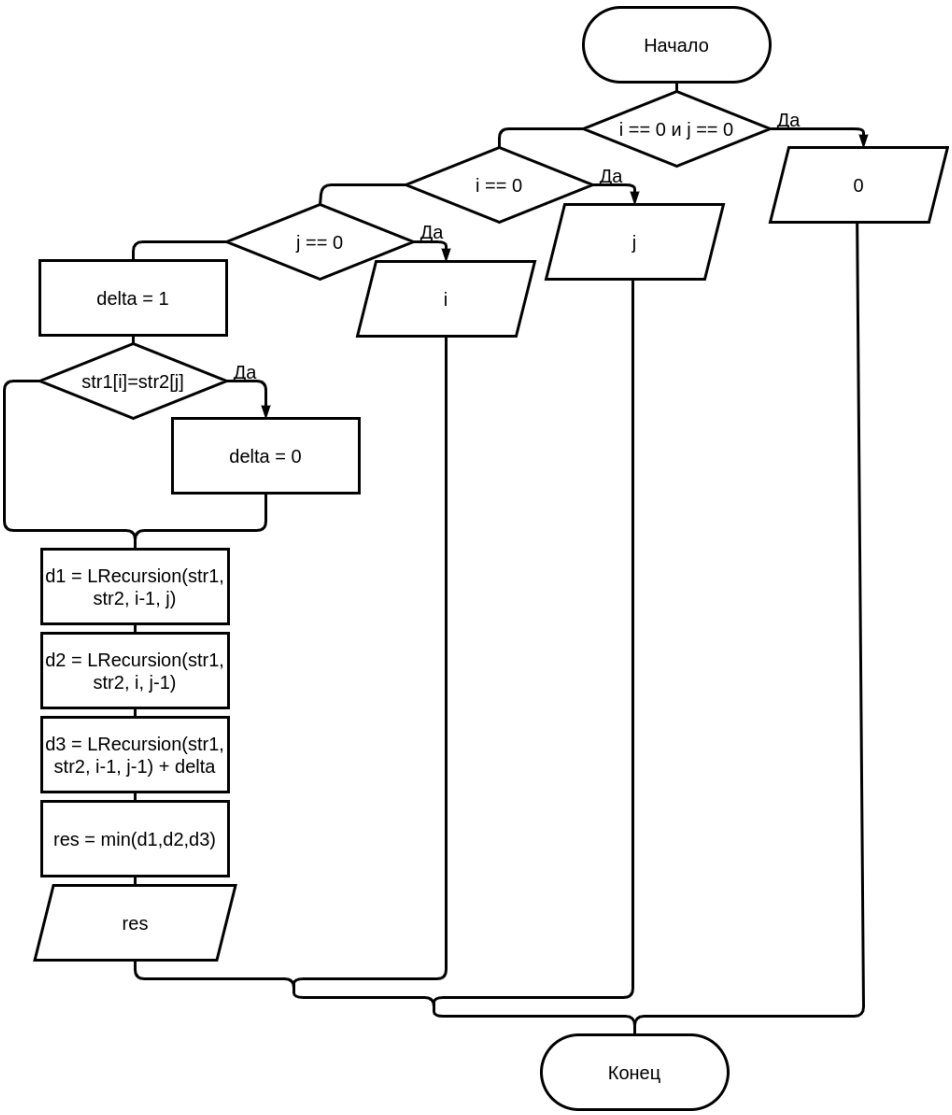

Рис. 2.2: Схема алгоритма нахождения расстояния Левенштейна рекурсивным способом

2.1.3 Схема рекурсивного алгоритма поиска расстояния Левенштейна с кешем

На рисунке 2.3 показана схема алгоритма расчета расстояния Левенштейна с помощью рекурсии с кешем.

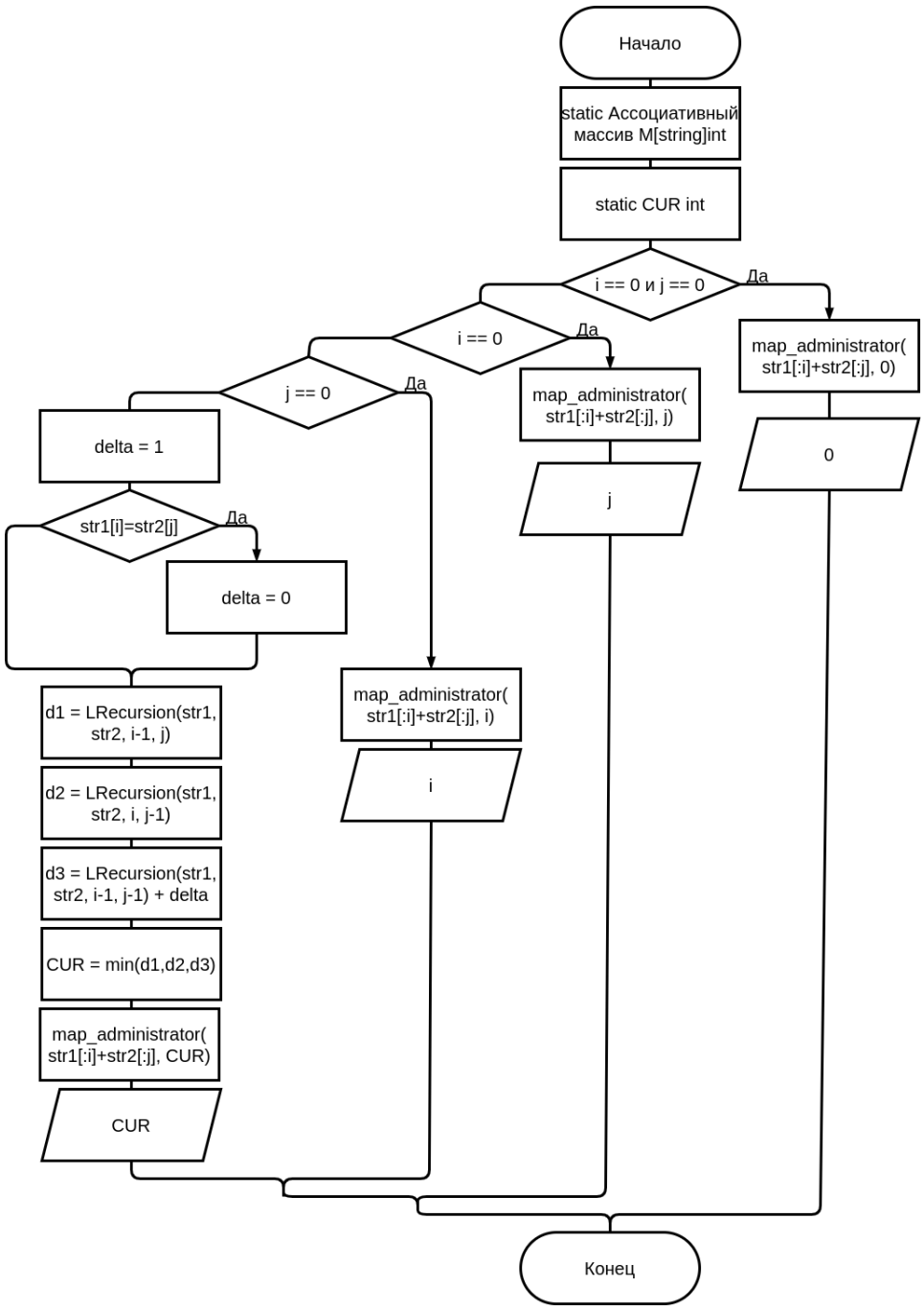

Рис. 2.3: Схема алгоритма нахождения расстояния Левенштейна рекурсивным способом с кешем

2.1.4 Схема рекурсивного алгоритма поиска расстояния Дамерау - Левенштейна

На рисунке 2.4 показана схема алгоритма расчета расстояния Левенштейна с помощью рекурсии.

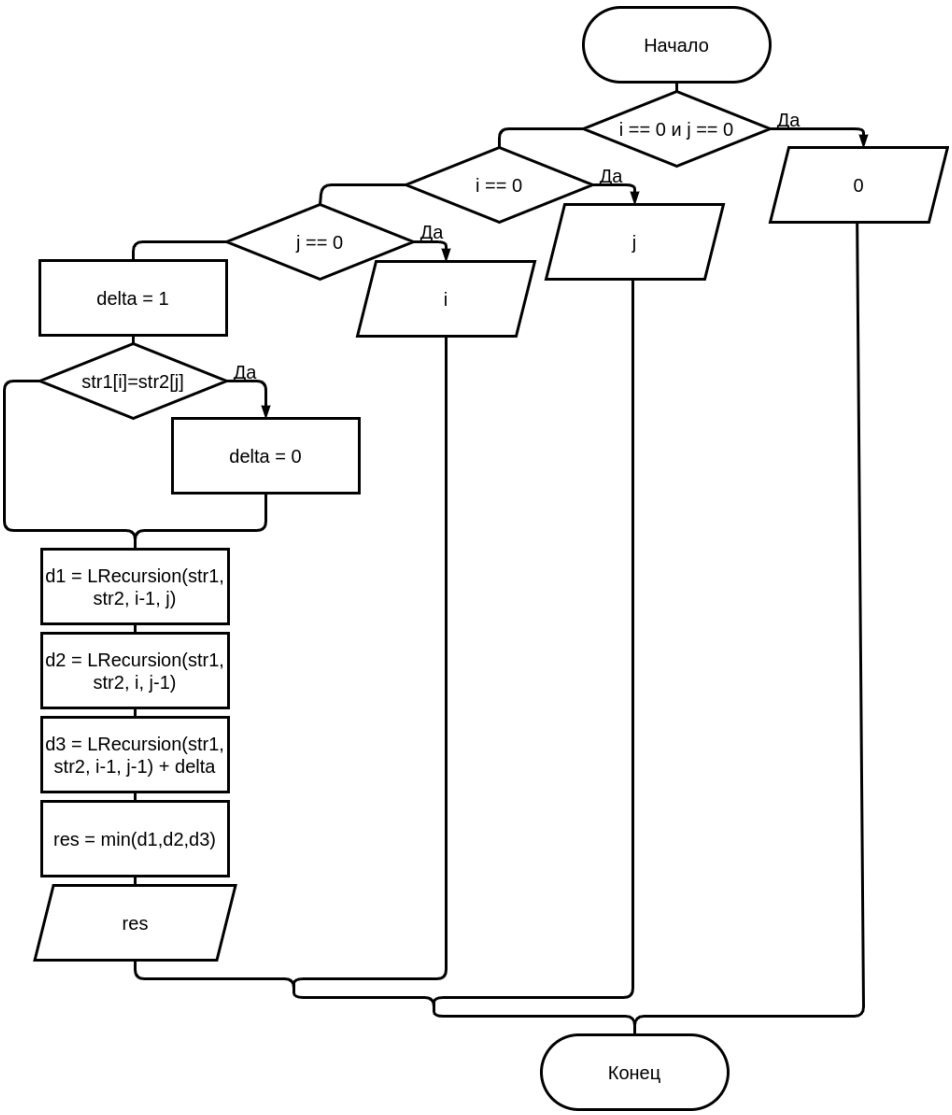

Рис. 2.4: Схема алгоритма нахождения расстояния Левенштейна рекурсивным способом

2.2 Структуры данных

При реализации приведенных алгоритмов потребуются следующие типы данных: матрица, специализированная матрица для нахождения расстояния Левенштейна, ассоциативный массив.

2.3 Тестирование

2.3.1 Классы эквивалентности

Для алгоритма Левенштейна можно выделить следующие классы эквивалентности:

1. По длине строки
   (a) Строки одинаковой длины
   (b) Строки разной длины
2. По количеству одинаковых символов
   (a) Одна строка входит в другую
   (b) В строках есть повторяющиеся части
   (c) В строках нет повторяющихся символов

Для алгоритма Дамерау-Левенштейна можно выделить следующие классы эквивалентности.

1. По длине строки
   (a) Строки одинаковой длины
   (b) Строки разной длины
2. По количеству одинаковых символов
   (a) Одна строка входит в другую

(b) В строках есть повторяющиеся части

(c) В строках нет повторяющихся символов

3. По наличию опечаток

(a) В строке есть опечатка (пе->еп)

(b) В строке опчатки нет (пе->еп)

2.3.2 Способы тестирования

При разработке программы удобно использовать следующие методы тестирования:

1. Модульные тесты

2. Функциональные тесты

2.4 Вывод конструкторской части

На основе данных, полученных в аналитическом разделе, были построены схемы используемых алгоритмов, выделены необходимые для реализации структуры данных и методы тестирования.

3. Технологическая часть

3.1 Требования к ПО

Требования к программному обеспечению:

1. Наличие меню для реализации выбора пользователя.
2. На вход подаются 2 строки
3. Результат в зависимости от выбора пользователя:
   (a) два числа - расстояние Левенштейна между строками и расстояние Дамерау-Левенштейна между строками
   (b) время, затраченное на обработку строк каждым из исследуемых алгоритмов.

3.2 Выбор языка программирования

Был выбран язык Go для знакомства с ним.

3.3 Структуры данных

На листинге 3.3 представлено описание структуры матрицы.

Листинг 3.1: Структура матрицы

```
type Matrix struct {
    matrix [][]int
    rows int
    columns int
}
```

На листинге 3.2 представлено описание структуры специализированной матрицы для нахождения расстояния Левенштейна.

Листинг 3.2: Структура специализированной матрицы Левенштейна

```
type MatrixLeventein struct {
    matrix Matrix
    x string // horisontal
    y string // vertical
}
```

Структура данных ассоциативный массив представлена в языке Go.

3.4 Реализация алгоритмов

На листинге 3.3 представлена реализация алгоритмов.

Листинг 3.3: Реализация матричного алгоритма поиска расстояния Левенштейна

```
func get_levenshtein_matrix(str1 string, str2 string) Matrix{
    var m Matrix = make_empty_matrix(len(str1), len(str2))   //str1 -
       vertical word, str2 - horisontal word
    for row := 0; row < len(str1); row++{
        for column := 0; column < len(str2); column++{
            if (row == 0 && column == 0){
                m.matrix[row][column] = 0
            } else if (row == 0) {
                m.matrix[row][column] = m.matrix[row][column - 1] + 1
            } else if (column == 0) {
                m.matrix[row][column] = m.matrix[row - 1][column] + 1
            } else {
                var delta int = 1 // for diag step
                if (str1[row] == str2[column]){
                    delta = 0
                }

                var con1 int = m.matrix[row - 1][column] + 1 // vertical
                   step
```

```
18                  var con2 int = m.matrix[row][column - 1] + 1 // horisontal
                        step
19                  var con3 int = m.matrix[row - 1][column - 1] + delta// diag
                        step
20
21                  m.matrix[row][column] = min3(con1, con2, con3)
22              }
23          }
24      }
25      return m
26  }
```

На листинге 3.4 представлена реализация алгоритмов.

Листинг 3.4: Реализация рекурсивного алгоритма поиска расстояния Левенштейна

```
1  func LRecursion(s1 *string, s2 *string, i int, j int) int{
2    if (i == 0 && j == 0){
3      return 0
4    } else if (i == 0 ){
5      return j
6    } else if (j == 0){
7      return i
8    } else if (i > 0 && j > 0){
9      var delta int = 1
10     if ((*s1)[i] == (*s2)[j]){
11       delta = 0
12     }
13     return min3(LRecursion(s1, s2, i, j - 1) + 1, LRecursion(s1, s2, i - 1,
           j) + 1, LRecursion(s1, s2, i - 1, j -1) + delta)
14   }
15   return -1
16 }
```

На листинге 3.5 представлена реализация алгоритмов.

Листинг 3.5: Реализация рекурсивного алгоритма с кешем поиска расстояния Левенштейна

```
1  func getLRecursionKesh()(func (s1 *string, s2 *string, i int, j int) int){
```

```
  var delta int = 1
  var ok bool
  var Cur int
  var M map[string] int = make(map[string]int)
  var map_administrator = func (s string, ans int){
    var v int
    var ok bool
    v, ok = M[s]
    if (!ok){
      M[s] = ans
    } else if (v > ans){
      M[s] = ans
    }
  }
  return func (s1 *string, s2 *string, i int, j int) int{
    if (i == 0 && j == 0){
      map_administrator(((*s1)[:i] + (*s2)[:j]), 0)
      return 0
    } else if (i == 0 ){
      map_administrator(((*s1)[:i] + (*s2)[:j]), j)
      return j
    } else if (j == 0){
      map_administrator(((*s1)[:i] + (*s2)[:j]), i)
      return i
    } else if (i > 0 && j > 0){
      if ((*s1)[i] == (*s2)[j]){
        delta = 0
      }
      Cur, ok = M[(*s1)[:i]+ (*s2)[:j]]
      if (!ok){
        Cur =  min3(DLRecursion(s1, s2, i, j - 1) + 1, DLRecursion(s1, s2,
           i - 1, j) + 1, DLRecursion(s1, s2, i - 1, j -1) + delta)
        map_administrator((*s1)[:i]+ (*s2)[:j], Cur)
      }
      return Cur
    }
    return -1
```

```
38    }
39  }
```

На листинге 3.6 представлена реализация алгоритмов.

Листинг 3.6: Реализация рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна

```
1  func DLRecursion(s1 *string, s2 *string, i int, j int) int{
2    if (i == 0 && j == 0){
3      return 0
4    } else if (i == 0 ){
5      return j
6    } else if (j == 0){
7      return i
8    } else if (i > 0 && j > 0 && (*s1)[i] == (*s2)[j - 1] && (*s1)[i - 1] ==
       (*s2)[j]){
9      var delta int = 1
10     if ((*s1)[i] == (*s2)[j]){
11       delta = 0
12     }
13     return min4(DLRecursion(s1, s2, i, j - 1) + 1, DLRecursion(s1, s2, i -
         1, j) + 1, DLRecursion(s1, s2, i - 1, j - 1) + delta,
14           DLRecursion(s1, s2, i - 2, j - 2) + delta)
15   } else if (i > 0 && j > 0){
16     var delta int = 1
17     if ((*s1)[i] == (*s2)[j]){
18       delta = 0
19     }
20     return min3(DLRecursion(s1, s2, i, j - 1) + 1, DLRecursion(s1, s2, i -
         1, j) + 1, DLRecursion(s1, s2, i - 1, j -1) + delta)
21   }
22   //fmt.Println("DLRECURSION: Impossible critical ERROR")
23   return -1
24 }
```

3.5 Тестирование

Модульные тесты

Таблица 3.1 – Тесты поиска расстояния Левенштейна

№	Строка 1	Строка 2	Вывод
1		slovo	5
2	no	slovo	4
3	olovo	slovo	1
4	slovo	slovo	0
5	sloov	slovo	2
6	oslovovo	slovo	3
7	net	slovo	5

Таблица 3.2 – Тесты поиска расстояния Дамерау-Левенштейна

№	Строка 1	Строка 2	Вывод
1		slovo	5
2	no	slovo	4
3	olovo	slovo	1
4	slovo	slovo	0
5	sloov	slovo	1
6	oslovovo	slovo	3
7	net	slovo	5

3.6 Вывод технологической части

Были реализованы исследуемые алгоритмы, программа прошла тесты и удовлетворяет требованиям.

4. Экспериментальная часть

Оценка качества работы алгоритмов. Экспериментальное сравнение работы различных алгоритмов поиска расстояния Левейнштейна (зависимость времени выполнения от длины слова).

4.1 Примеры работы

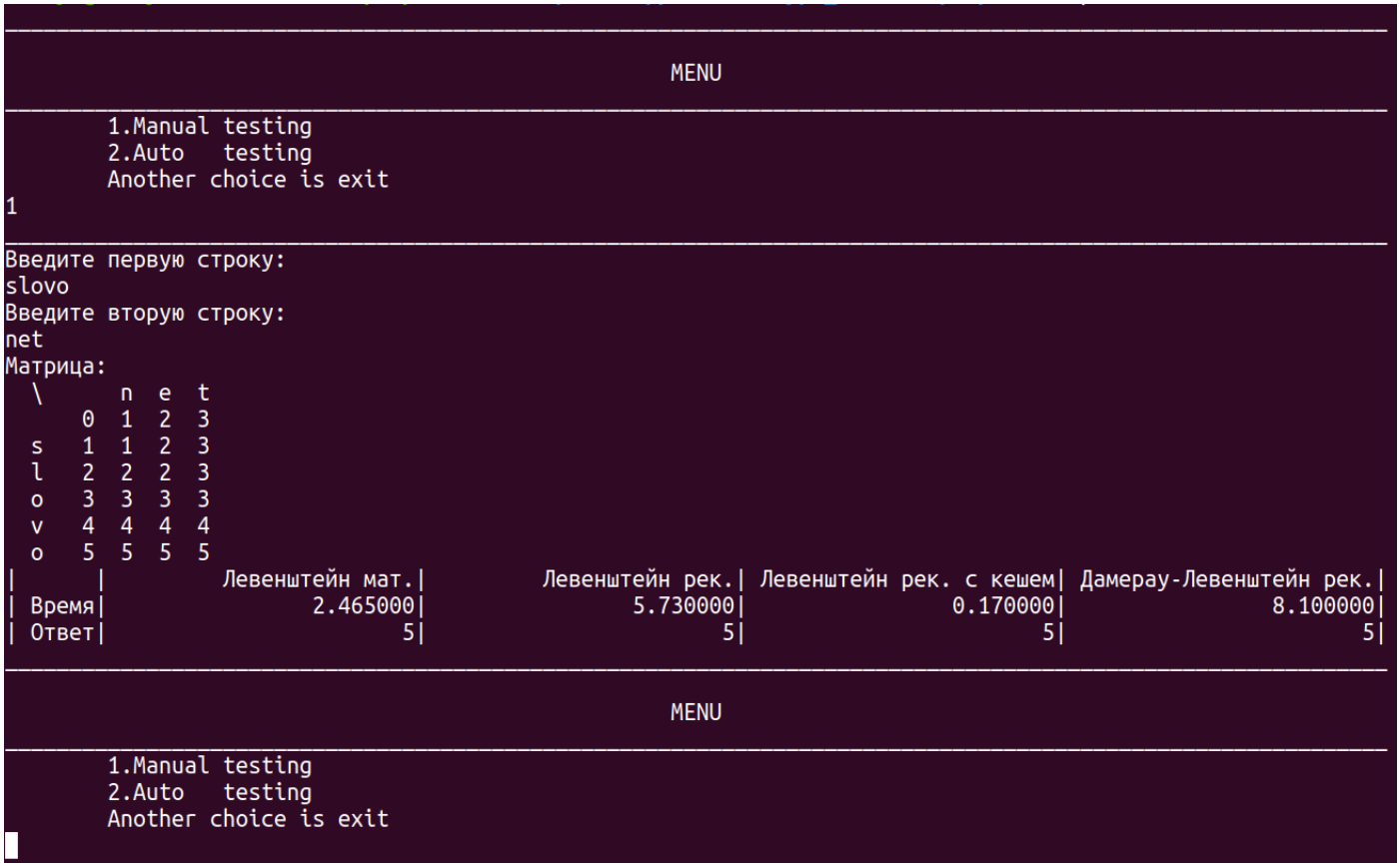

Рис. 4.1: Ручное тестирование

```
                                        MENU
        1.Manual testing
        2.Auto   testing
        Another choice is exit
2
| Длина|          Левенштейн мат.|          Левенштейн рек.| Левенштейн рек. с кешем| Дамерау-Левенштейн рек.|
|    10|                  0.215000|                 0.010000|                 0.055000|                 0.015000|
|    20|                  0.415000|                 0.035000|                 0.110000|                 0.050000|
|    30|                  0.455000|                 0.195000|                 0.145000|                 0.150000|
|    40|                  0.850000|                 0.655000|                 0.110000|                 1.240000|
|    50|                  1.140000|                 4.625000|                 0.150000|                 6.375000|
|    60|                  1.170000|                21.690000|                 0.160000|                17.445000|
|    70|                  1.660000|                87.630000|                 0.740000|               162.710000|
|    80|                  2.025000|               594.445000|                 4.390000|               894.820000|
|    90|                  6.380000|              3897.640000|                23.975000|              4958.235000|
|   100|                  1.990000|             16403.745000|                80.670000|             20156.895000|
                                        MENU
        1.Manual testing
        2.Auto   testing
        Another choice is exit
```

Рис. 4.2: Автоматическое тестирование

4.2 Замеры времени

На рисунках 4.3 и 4.4 показаны графические результаты сравнения исследуемых алгоритмов по времени.

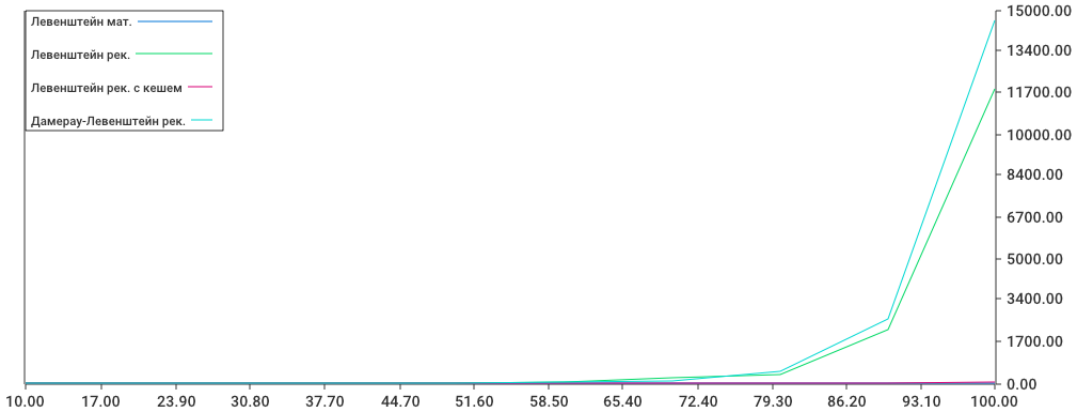

Рис. 4.3: Сравнение всех 4 исследуемых алгоритмов по времени

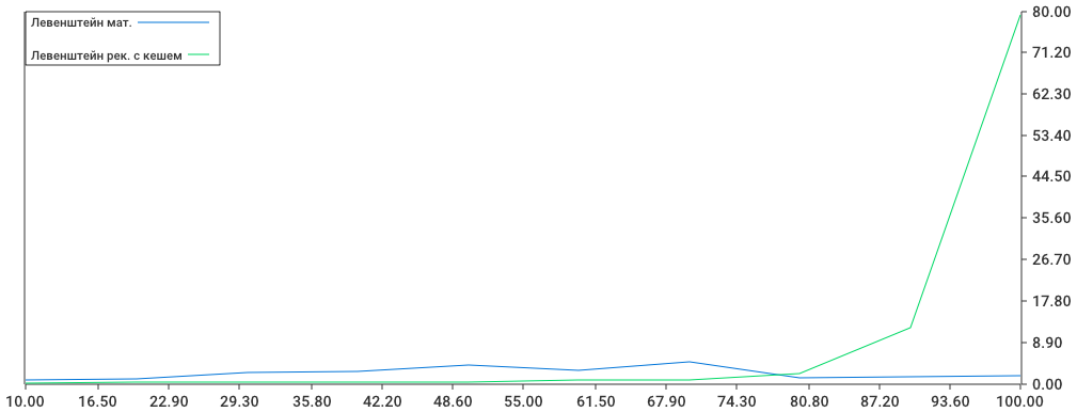

Рис. 4.4: Сравнение матричного и рекурсивного с кешем алгоритмов поиска расстояния Левенштейна

4.3 Вывод экспериментальной части

Таким образом, по графикам подтвердилось предположение, что самый эффективный алгоритм поиска расстояния Левенштейна из рассматриваемых в работе – рекурсивный метод с кешем. Рекурсивный алгоритм поиска расстояния Дамерау - Левенштейна дольше, чем рекурсивный алгоритм Левенштейна, что объясняется дополнительными операциями поиска перепутанных букв: (fd -> df).

5. Заключение

В данной работе был проведен обзор алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна. Изучены алгоритмы поиска расстояния между строками: рекурсивные алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна, рекурсивный алгоритм с кешем поиска расстояния Левенштейна, матричный алгоритм поиска Левенштейна. Получены практические навыки реализации исследуемых алгоритмов на языке программирования Go. Проведён сравнительный анализ алгоритмов по затрачиваемым ресурсам (зависимость времени от длины массива). Экспериментально подтверждены различия в эффективности алгоритмов с указанием лучших и худших случаев. При сравнении данных алгоритмов получены следующие результаты: самый эффективный алгоритм поиска расстояния Левенштейна из рассматриваемых в работе - рекурсивный алгоритм с кешем.

Список литературы

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

[2] Gratzer George A. More Math Into LaTeX. 4th изд. Boston: Birkhauser, 2007.

[3] The Haskell purely functional programming language [Электронный ресурс]. Режим доступа: https://haskell.org/. Дата обращения: 16.09.2020.

[4] News, Status Discussion about Stadard C++ [Электронный ресурс]. Режим доступа: https://isocpp.org. Дата обращения: 20.09.2020.

[5] Debian – универсальная операционная система [Электронный ресурс]. Режим доступа: https://www.debian.org/. Дата обращения: 20.09.2020.

[6] Linux – Getting Started [Электронный ресурс]. Режим доступа: https://linux.org. Дата обращения: 20.09.2020.

[7] Процессор Intel® Core™ i5-3550 [Электронный ресурс]. Режим доступа: https://ark.intel.com/content/www/ru/ru/ark/products/65516/intel-core-i5-3550-processor-6m-cache-up-to-3-70-ghz.html. Дата обращения: 20.09.2020.

[8] Profiling – Haskell profiling [Электронный ресурс]. Режим доступа: https://downloads.haskell.org/ghc/8.8.1/docs/html/user$_g$*uide*/*index.html*. : 20.09.2020.