МГТУ им. Баумана

Лабораторная работа №1

По курсу: "Анализ алгоритмов"

Расстояние Левенштейна

Работу выполнил: Левушкин Илья, ИУ7-52Б

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

Москва, 2019

Оглавление

Введение **2**

1 Аналитическая часть **4**
1.1 Цель 4
1.2 Задачи 4
1.3 Алгоритмы Левенштейна и Дамерау-Левенштейна 5
1.4 Выводы 6

2 Конструкторская часть **7**
2.1 Способы реализаций алгоритмов 7
2.2 Схемы алгоритмов 9
2.3 Требования к программе 17
2.4 Выводы 17

3 Технологическая часть **18**
3.1 Выбор ЯП 18
3.2 Замер времени 18
3.3 Сведения о модулях программы 18
3.4 Тесты 23
3.5 Выводы 31

4 Исследовательская часть **32**
4.1 Замер времени 32
4.2 Выводы 34

Заключение **35**

Введение

Алгоритмы нечеткого поиска (также известного как поиск по сходству или fuzzy string search) являются основой систем проверки орфографии и полноценных поисковых систем вроде Google или Yandex, а также применяются в биоинформатике для сравнения генов, хромосом и белков. Например, такие алгоритмы используются для функций наподобие «Возможно вы имели в виду ...» в тех же поисковых системах [2].

Нечеткий поиск является крайне полезной функцией любой поисковой системы. Вместе с тем, его эффективная реализация намного сложнее, чем реализация простого поиска по точному совпадению.

Задачу нечеткого поиска можно сформулировать так: «По заданному слову найти в тексте или словаре размера n все слова, совпадающие с этим словом (или начинающиеся с этого слова) с учетом k возможных различий».

Например, при запросе «Машина» с учетом двух возможных ошибок, найти слова «Машинка», «Махина», «Малина», «Калина» и так далее.

Алгоритмы нечеткого поиска характеризуются метрикой — функцией расстояния между двумя словами, позволяющей оценить степень их сходства в данном контексте. Строгое математическое определение метрики включает в себя необходимость соответствия условию неравенства треугольника (X — множество слов, p — метрика):

$$\rho(x, y) \leqslant \rho(x, z) + \rho(z, y), \quad x, y, z \in X \tag{1}$$

Между тем, в большинстве случаев под метрикой подразумевается более общее понятие, не требующее выполнения такого условия, это понятие можно также назвать расстоянием.

В числе наиболее известных метрик — расстояния Хемминга, Левенштейна и Дамерау-Левенштейна. При этом расстояние Хемминга является метрикой только на множестве слов одинаковой длины, что сильно ограничивает область его применения.

Впрочем, на практике расстояние Хемминга оказывается практически бесполезным, уступая более естественным с точки зрения человека метрикам. Поэтому больший интерес для изучения представляет собой алгоритм Левенштейна, а также его модификация - Дамерау-Левенштейна, который учитывает распространенную ошибку человека при наборе слов: "Персетановка билз лежаищх бкув в солве месатми"

1 | Аналитическая часть

1.1 Цель

Целью данной лабораторной работы является изучение алгоритмов Левенштейна и Дамерау-Левенштейна.

1.2 Задачи

Для достижения поставленной цели необходимо решить следующие **задачи**:

- проанализировать и разработать существующие реализации алгоритмов Левенштейна и Дамерау-Левенштейна;
- выбрать технологии для последующих реализаций и иследования алгоритмов;
- реализовать эти алгоритмы;
- произвести тестирование корректности работы реализаций;
- сравнить быстродействие реализаций;
- описать и обосновать полученные результаты в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

1.3 Алгоритмы Левенштейна и Дамерау-Левенштейна

Задача по нахождению расстояния Левенштейна заключается в поиске минимального количества операций вставки/удаления/замены для превращения одной строки в другую.

При нахождении расстояния Дамерау — Левенштейна добавляется операция транспозиции (перестановки соседних символов).

Действия обозначаются так:

1. D (англ. delete) — удалить,
2. I (англ. insert) — вставить,
3. R (replace) — заменить,
4. M (match) - совпадение.

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:

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

где $m(a, b)$ равна нулю, если $a = b$ и единице в противном случае; $min\{\, a, b, c\}$ возвращает наименьший из аргументов [3], [5].

Расстояние Дамерау-Левенштейна вычисляется по следующей рекуррентной формуле:

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ min(& \\ D(i, j-1) + 1, & \\ D(i-1, j) + 1, & j > 0, i > 0 \\ D(i-1, j-1) + m(S_1[i], S_2[j]) & \\ D(i-2, j-2) + 1, & \text{if } i, j > 1 \text{ and } a_i = b_{j-1}, a_{i-1} = b_j \\ ) & otherwise \\ min(& \\ D(i, j-1) + 1, & \\ D(i-1, j) + 1, & j > 0, i > 0 \\ D(i-1, j-1) + m(S_1[i], S_2[j]) & \\ ) & \end{cases} \tag{1.2}$$

[4]

1.4 Выводы

В рамках данной работы были выбраны для исследования алгоритмы Левенштейна и Дамерау-Левенштейна. Математическое описание обоих алгоритмов приведено в формулах 1.1, 1.2

2 | Конструкторская часть

2.1 Способы реализаций алгоритмов

Алгоритмы Левенштейна и Дамерау-Левенштейна могут быть реализованы следущими способами:

- рекурсивный метод;
- матричный метод.

Первый представляет собой рекурсивное вычисление всех расстояний между словами, и выбором минимального из полученных решений.

Сами рекурсии работ алгоритмов задаются следующим образом:

Обозначения:

- str1, str2 - Строки
- p - Расстояние
- length - Длина Строки
- str[length - 1] - Строка без первого элемента слева
- == - Проверка на равенство

Рекурсия алгоритма Левенштейна

На входе: str1, str2

- При первом вызове рекурсии расстояние p = 0

- Если длина одной из строк стала равна 0, то к полученному ранее расстоянию р добавляется длина другой строки. (см. формулу 1.1)
- В случае, если длины обоих строк равны 0, то к полученному ранее расстоянию р ничего не добавляется.
- Вызов рекурсий со следующими параметрами:
  - $str1[length - 1]$, $str2$, $p + 1$
  - $str1$, $str2[length - 1]$, $p + 1$
  - $str1[length - 1]$, $str2[length - 1]$, $p+$ если $str1[1] == str2[1]$, то 0, иначе 1

Рекурсия алгоритма Дамерау-Левенштейна
На входе: str1, str2, p

- При первом вызове рекурсии р = 0
- Если длина одной из строк стала равна 0, то к полученному ранее расстоянию р добавляется длина другой строки. (см. формулу 1.1)
- В случае, если длины обоих строк равны 0, то к полученному ранее расстоянию р ничего не добавляется.
- Вызов рекурсий со следующими параметрами:
  - $str1[length - 1]$, $str2$, $p + 1$
  - $str1$, $str2[length - 1]$, $p + 1$
  - $str1[length - 1]$, $str2[length - 1]$, $p+$если $str1[1] == str2[1]$, то 0, иначе 1
  - если $str1[1] == str2[2]$ и $str1[2] == str1[1]$, то $str1[length - 2]$, $str2[length - 2]$, $p + 1$, иначе не вызывать рекурсию.

Полученные "листья дерева"и будут являться решениями.

Матричная реализация алгоритмов Матричная реализация алгоритмов редставляет собой нахождение матрицы наименьших расстояний слов:

$$\begin{pmatrix} p(str1[1], str2[1]) & p(str1[1,2], str2[1]) & \dots & p(str[length], str2[1]) \\ p(str1[1], str2[1,2]) & p(str1[1,2], str2[1,2]) & \dots & p(str[length], str2[1,2]) \\ \vdots & \vdots & \ddots & \vdots \\ p(str1[1], str2[]length]) & p(str1[1,2], str2[length] & \dots & p(str[length], str2[length]) \end{pmatrix} \tag{2.1}$$

Где последнее расстояние слов $(str1[length], str2[length])$ и будет являться решением.

Каждая элемент матрицы находится по следующим правилам:

Алгоритм Левенштейна

$$p[i][j] = min(p[i-1][j] + 1, p[i][j-1] + 1, p[i-1][j-1] + k) \tag{2.2}$$

Где $k = 0$, если $str1[i] == str2[j]$, иначе $k = 1$

Алгоритм Дамерау-Левенштейна

$$p[i][j] = min(p[i-1][j]+1, p[i][j-1]+1, p[i-1][j-1]+k, p[i-2][j-2]+f) \tag{2.3}$$

Где $k = 0$, если $str1[i] == str2[j]$, иначе $k = 1$

$f = 1$, если $str1[i] == str2[j-1] \& str1[i-1] == str2[j]$, иначе $f = 2$

2.2 Схемы алгоритмов

На рисунках 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7 представлены схемы реализаций алгоритмов Левенштейна и Дамерау-Левенштейна.

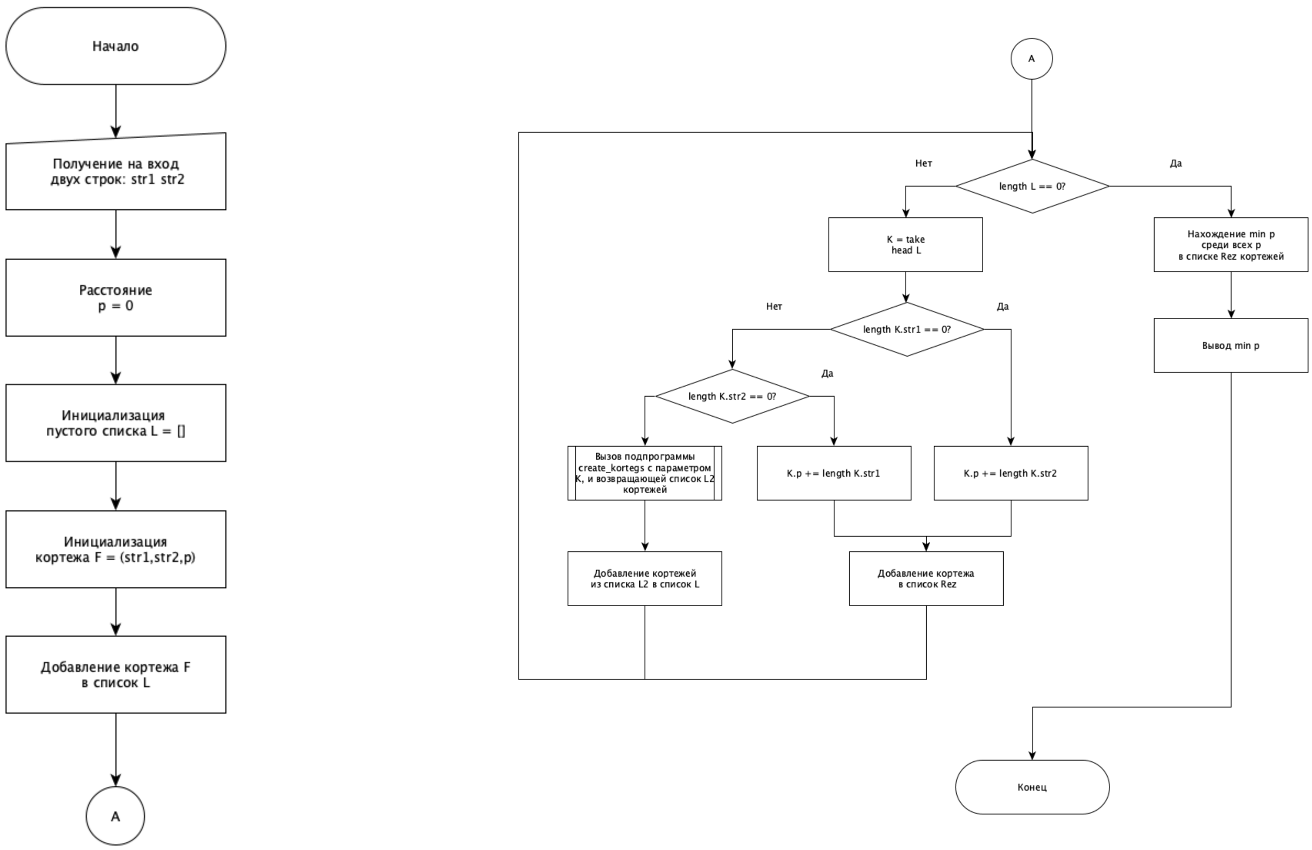


Рис. 2.1: Рекурсивная реализация.

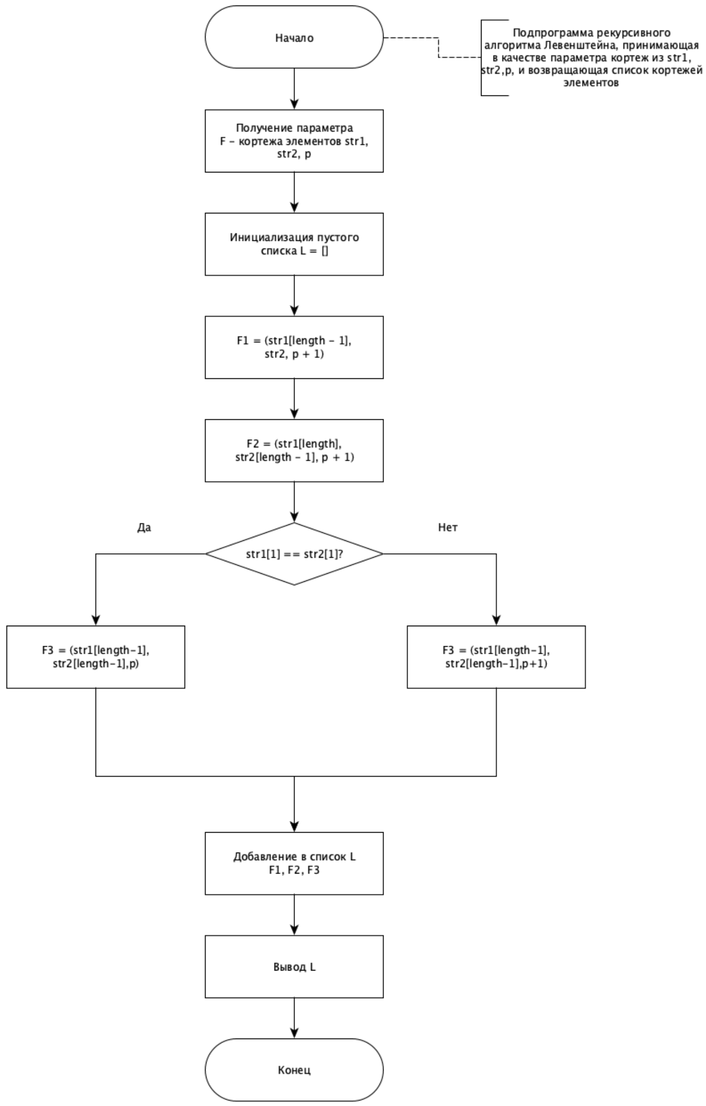


Рис. 2.2: Подпрограмма алгоритма Левенштейна.

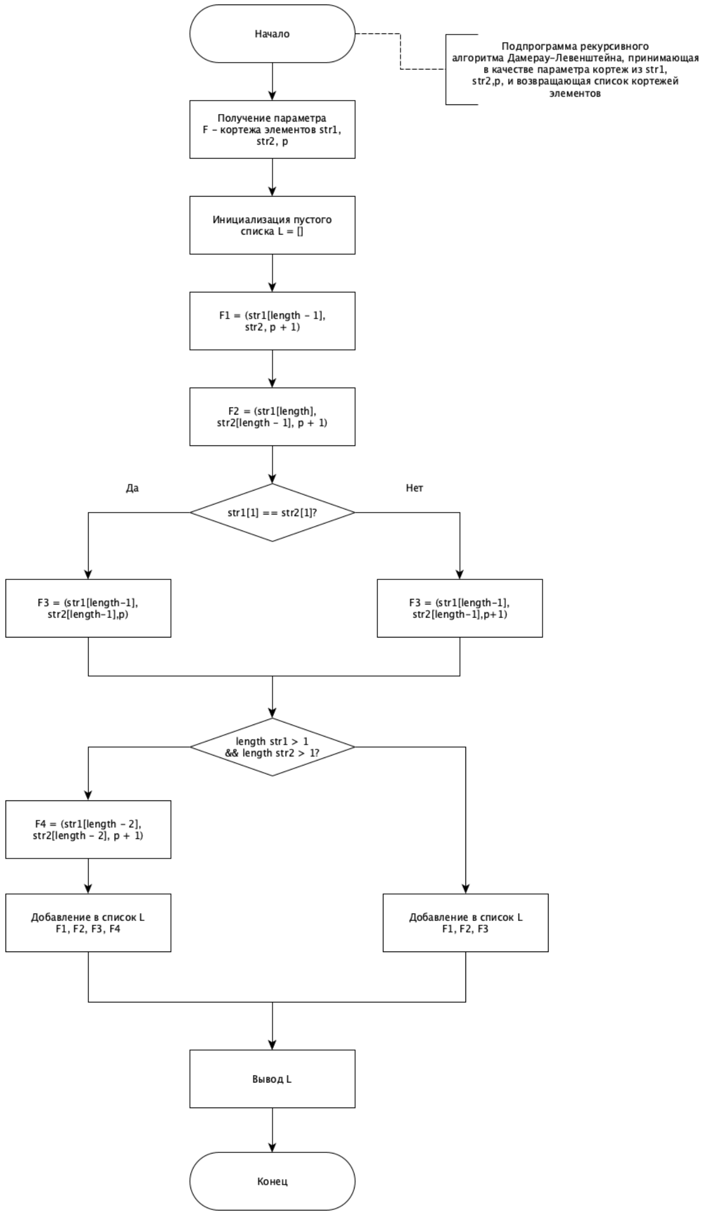


Рис. 2.3: Подпрограмма алгоритма Дамерау-Левенштейна.

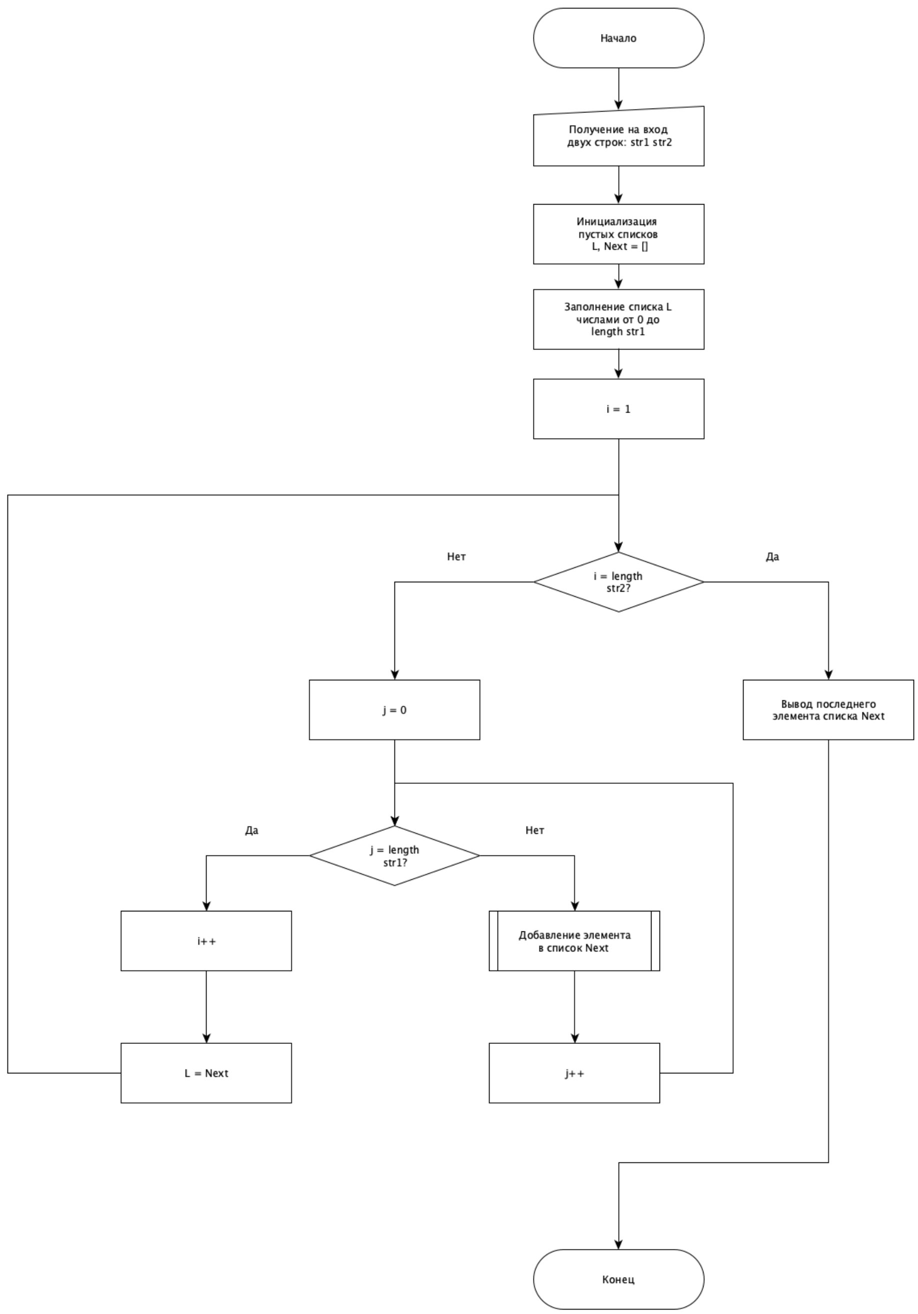


Рис. 2.4: Матричный алгоритм Левенштейна.

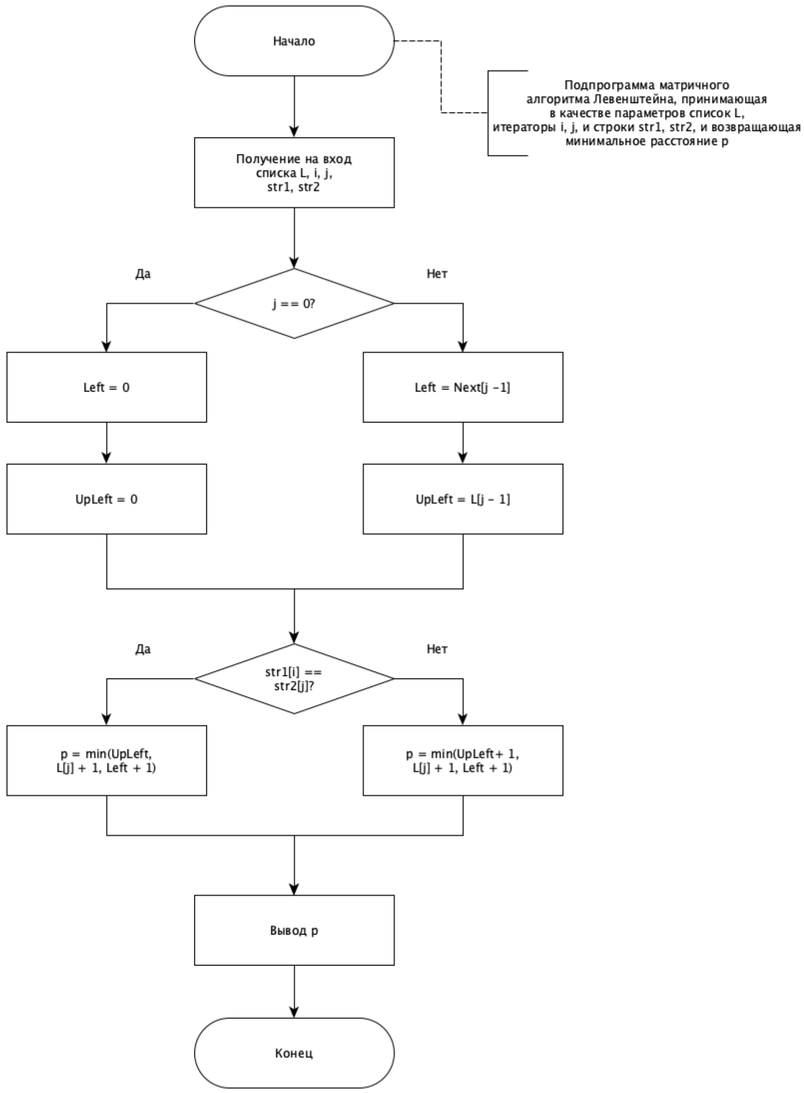


Рис. 2.5: Подпрограмма алгоритма Левенштейна.

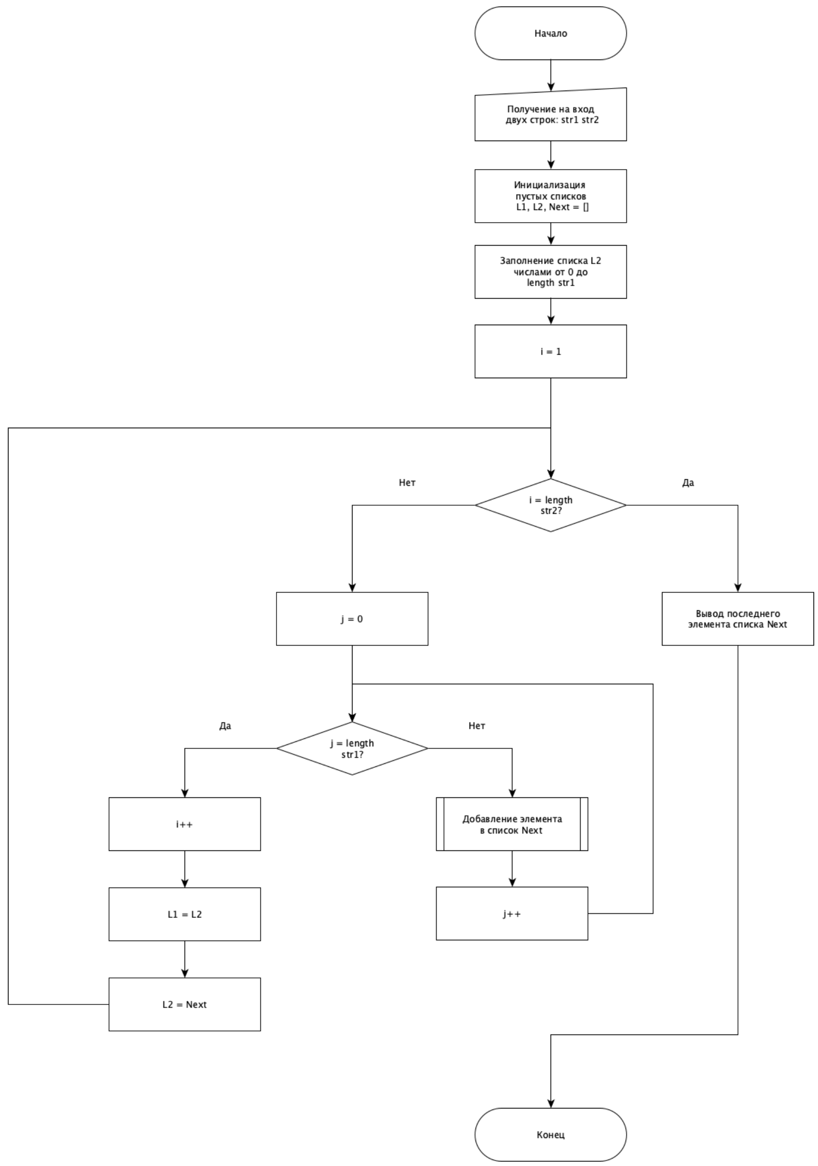


Рис. 2.6: Матричный алгоритм Дамерау-Левенштейна.

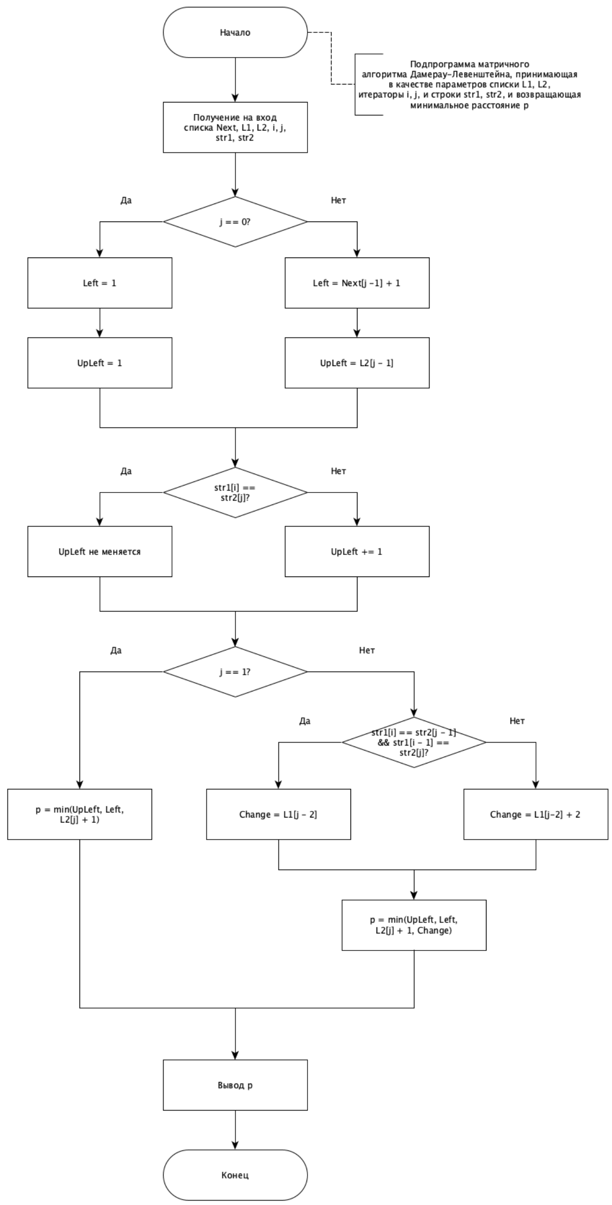


Рис. 2.7: Подпрограмма алгоритма Дамерау-Левенштейна.

2.3 Требования к программе

- Программа должна предоставлять доступный и понятный интерфейс с выбором алгоритмов (и всех их реализаций);
- Должно быть 2 матричных реализации каждого алгоритма:
  - хранит только те строки, которые необходимы для расчета последующих строк матрицы;
  - хранит всю матрицу целиком.
- На вход каждой функции, реализующей конкретный алгоритм, подаются две строки латинского или русского алфавита;
- Пользователь должен иметь возможность вводить строки вручную либо выбрать пункт случайного заполнения, в случае чего ему будет предоставлен выбор длины строк;
- Uppercase и Lowercase буквы считаются разными;
- Требования к выводу программы:
  - программа должна выдавать корректное минимальное расстояние между двумя словами (см. Введение);
  - при вводе пустых строк - программа не должна аварийно завершаться, а выдавать значение 0;
  - помимо полученного значения, программа должна выдать процессорное время, затраченное на время выполнения данной реализации алгоритма;
  - В случае выбора реализации матричного метода с запоминанием всей матрицы одного из алгоритмов, программа должна вывести на экран матрицу целиком.

2.4 Выводы

В результате проведенной работы было решено реализовать алгоритмы Левенштейна и Дамерау-Левенштейна рекурсивными и матричными способами; были разработаны схемы алгоритмов.

3 | Технологическая часть

3.1 Выбор ЯП

В качестве языка программирования был выбран **Haskell**, так как он является чистым функциональным языком и хорошо подходит для реализации рекурсивных алгоритмов. [1]

3.2 Замер времени

Время работы алгоритмов было замерено с помощью функции **getCPUTime** из библиотеки **System.CPUTime**.

3.3 Сведения о модулях программы

Программа состоит из:

- Main.hs - Главный файл программы, в котором располагается меню;
- Lib.hs - файл с подключенными модулями реализаций алгоритмов;
- Domerau_Levenshtein.hs - Файл с реализацией рекурсивного алгоритма Дамерау-Левенштейна;
- Levenshtein_matrix.hs - Файл с реализацией матричного алгоритма Левенштейна;
- Domerau_Levenshtein_matrix.hs - Файл с реализацией матричного алгоритма Дамерау-Левенштейна;

- Output_Levenshtein_matrix.hs - Файл с реализацией матричного алгоритма Левенштейна с хранением полной матрицы;
- Output_Domerau_Levenshtein_matrix.hs - Файл с реализацией матричного алгоритма Дамерау-Левенштейна с хранением полной матрицы;
- Spec.hs - Файл с модульными тестами.

Ниже приведены листинги 3.1, 3.2, 3.3, 3.4, 3.5 функций программы.

Листинг 3.1: Функция нахождения расстояния Дамерау-Левенштейна рекурсивно

```
import Data.List

f ([],          [],          k) = [k]
f (xs,          [],          k) = [k + length xs]
f ([],          ys,          k) = [k + length ys]
f (xs@(x:xs'), ys@(y:ys'), k) | ((xs' /= []) && (ys' /=
    [])) =
calc_list $ (xs', ys, k + 1)
: (xs, ys', k + 1)
: (xs', ys', k + if x == y then 0 else 1)
: if ((x == head ys') && (y == head xs'))
then [(tail xs', tail ys', k + 1)] else []
| otherwise                         =
calc_list $ (xs', ys, k + 1)
: (xs, ys', k + 1)
: [(xs', ys', k + if x == y then 0 else 1)]
calc_list []     = []
calc_list (x:xs) = f x ++ calc_list xs

domerau_levenshtein s1 s2 = minimum $ calc_list [(s1,
   s2, 0)]
```

Листинг 3.2: Функция нахождения расстояния Левенштейна матрично

```
import Data.List

levenshtein s1 s2 = last $ foldl (transform s1)
```

```
[0..length s1] s2
where transform str xs@(x:xs') c = res where
res = x + 1 : zipWith4 compute str xs xs' res
compute c' x y z = minimum [y + 1
, z + 1
, x + if c' == c
then 0 else 1]
```

Листинг 3.3: Функция нахождения расстояния Дамерау-Левенштейна матрично

```
import Data.List


second_elem (_,x,_) = x

res str xs@(x : xs') p1  = (xs, result, p1)
where result            = x + 1 : zipWith4 compute str
   xs xs' result
compute c1 x y z = minimum [y + 1
, z + 1
, x + if c1 == p1 then 0 else 1]

res' str xs@(x : xs') ys p1 p2  = (xs, x + 1 : result,
   p1)
where result = (compute (head str)
(head xs)
(head xs')
(x + 1))
: zipWith6 compute_2
(tail str)
str
(tail xs)
(tail xs')
result ys
compute   c1 x  y z      = minimum [y + 1
, z + 1
, x + if c1 == p1
then 0 else 1]
compute_2 c1 c2 x y z k = minimum [y + 1
, z + 1
```

```
    , x + if c1 == p1
    then 0 else 1
    , k + if
    ((c1 == p2) && (c2 == p1))
    then 1 else 2]


    domerau_levenshtein s1 s2 = last $ second_elem (foldl (
       transform s1)
    ([], [0..(length s1)], ' ') s2)
    where transform str
    (ys, xs@(x : xs'), p2)
    p1 | ys == []  = res str xs p1
    | otherwise = res' str xs ys p1 p2
```

Листинг 3.4: Функция нахождения расстояния Левенштейна матрично с хранением матрицы

```
    import Data.List

    levenshtein s1 s2 = (last $ fst rezult, reverse $ snd
       rezult) where
    rezult = foldl (transform s1) (fill_list, [fill_list])
       s2
    where fill_list = [0..length s1]
    transform str (xs@(x:xs'), ys) c = (res, res : ys)
       where
    res = x + 1 : zipWith4 compute str xs xs' res
    compute c' x y z = minimum [y + 1
    , z + 1
    , x + if c' == c
    then 0 else 1]
```

Листинг 3.5: Функция нахождения расстояния Дамерау-Левенштейна матрично с хранением матрицы

```
    import Data.List


    second_elem (_,x,_) = x

```

```
res str xs@(x : xs') p1 zs = (xs, (result, result : zs), p1)
where result           = x + 1 : zipWith4 compute str xs xs' result
compute c1 x y z = minimum [y + 1
, z + 1
, x + if c1 == p1 then 0 else 1]

res' str xs@(x : xs') ys p1 p2 zs = (xs
, (x + 1 : result
, ((x + 1) : result) : zs)
, p1)
where result                   = (compute (head str)
(head xs)
(head xs')
(x + 1))
: zipWith6 compute_2 (tail str)
str
(tail xs)
(tail xs')
result ys
compute c1 x y z        = minimum [y + 1
, z + 1
, x + if c1 == p1 then 0 else 1]
compute_2 c1 c2 x y z k = minimum [y + 1
, z + 1
, x + if c1 == p1 then 0 else 1
, k + if ((c1 == p2) && (c2 == p1))
then 1 else 2]


domerau_levenshtein s1 s2 = (last $ fst rezult, reverse $ snd rezult) where
rezult = second_elem (foldl (transform s1)
([], (fill_list, [fill_list]), ' ')
s2)
where fill_list = [0..(length s1)]
transform str
(ys, (xs@(x : xs'), zs), p2)
p1 | ys == []  = res str xs p1 zs
```

```
    | otherwise = res' str xs ys p1 p2 zs
```

3.4 Тесты

Тестирование было организовано с помощью библиотеки **TestHUnit**. Было создано две вариации тестов: В первой сравнивались результаты функции с реальным результатом.

Во второй сравнивались реузультаты двух функций (рекурсивной и табличной). При сравнении результатов двух функций использовалась функция randomRs из библиотеки System.Random, которая генерирует случайную строку нужной длины.

Листинг 3.6: Функция генерации случайной строки длины 7

```
take 7 $ randomRs ('a', 'z') randGen
```

Где randGen - генератор , инициализированный ранее с помощью функции newStdGen.

В таблицах представлены тесты всех пяти реализаций алгоритмов:

- Рекурсивный алгоритм Дамерау-Левенштейна
- Матричный алгоритм Левенштейна
- Матричный алгоритм Дамерау-Левенштейна
- Матричный алгоритм Левенштейна (с выводом матрицы)
- Матричный алгоритм Дамерау-Левенштейна (с выводом матрицы)

Ниже приведены таблицы 3.1, 3.2, 3.3, 3.4, 3.5 тестов алгоритмов.

Таблица 3.1: Рекурсивный алгоритм Дамерау-Левенштейна.

Описание теста	Ввод		Ожидаемое	Вывод	Результат
	Строка №1	Строка №2			
Пустые строки	"_"	"_"	0	0	✓
Вторая строка пуста	"a"	"_"	1	1	✓
Первая строка пуста	"_"	"a"	1	1	✓
Одинаковые строки	"equal"	"equal"	0	0	✓
Входные строки в кириллице	"одинаковые"	"одинаковые"	0	0	✓
Обычный случай	"usual"	"specul"	5	5	✓
Случай с совпадением 1 символа	"usual"	"qswer"	4	4	✓
Одна строка больше другой	"usual"	"moreusual"	4	4	✓
Случай с перестановкой двух соседних символов	"mroesuua"	"moreusual"	3	3	✓

Таблица 3.2: Матричный алгоритм Левенштейна.

Описание теста	Ввод		Ожидаемое	Вывод	Результат
	Строка №1	Строка №2			
Пустые строки	"_"	"_"	0	0	✓
Вторая строка пуста	"a"	"_"	1	1	✓
Первая строка пуста	"_"	"a"	1	1	✓
Одинаковые строки	"equal"	"equal"	0	0	✓
Входные строки в кириллице	"одинаковые"	"одинаковые"	0	0	✓
Обычный случай	"usual"	"specul"	5	5	✓
Случай с совпадением 1 символа	"usual"	"qswer"	4	4	✓
Одна строка больше другой	"usual"	"moreusual"	4	4	✓

Таблица 3.3: Матричный алгоритм Домерау-Левенштейна.

Описание теста	Ввод		Ожидаемое	Вывод	Результат
	Строка №1	Строка №2			
Пустые строки	"_"	"_"	0	0	✓
Вторая строка пуста	"a"	"_"	1	1	✓
Первая строка пуста	"_"	"a"	1	1	✓
Одинаковые строки	"equal"	"equal"	0	0	✓
Входные строки в кириллице	"одинаковые"	"одинаковые"	0	0	✓
Обычный случай	"usual"	"specul"	5	5	✓
Случай с совпадением 1 символа	"usual"	"qswer"	4	4	✓
Одна строка больше другой	"usual"	"moreusual"	4	4	✓
Случай с перестановкой двух соседних символов	"mroesuua"	"moreusual"	3	3	✓

Таблица 3.4: Матричный алгоритм Левенштейна с выводом матрицы.

Описание теста	Ввод		Ожидаемое	Вывод	Рез
	Строка №1	Строка №2			
Пустые строки	"_"	"_"	$(0, [[0]])$	$(0, [[0]])$	✓
Вторая строка пуста	"a"	"_"	$(1, [[0, 1]])$	$(1, [[0, 1]])$	✓
Первая строка пуста	"_"	"a"	$(1, [[0], [1]])$	$(1, [[0], [1]])$	✓
Одинаковые строки	"equal"	"equal"	(0,[$[0, 1, 2, 3, 4, 5]$ $, [1, 0, 1, 2, 3, 4]$ $, [2, 1, 0, 1, 2, 3]$ $, [3, 2, 1, 0, 1, 2]$ $, [4, 3, 2, 1, 0, 1]$ $, [5, 4, 3, 2, 1, 0]$])	(0,[$[0, 1, 2, 3, 4, 5]$ $, [1, 0, 1, 2, 3, 4]$ $, [2, 1, 0, 1, 2, 3]$ $, [3, 2, 1, 0, 1, 2]$ $, [4, 3, 2, 1, 0, 1]$ $, [5, 4, 3, 2, 1, 0]$])	✓
Входные строки в кириллице	"одинаковые"	"одинаковые"	(0,[$[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$ $, [1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$ $, [2, 1, 0, 1, 2, 3, 4, 5, 6, 7, 8]$ $, [3, 2, 1, 0, 1, 2, 3, 4, 5, 6, 7]$ $, [4, 3, 2, 1, 0, 1, 2, 3, 4, 5, 6]$ $, [5, 4, 3, 2, 1, 0, 1, 2, 3, 4, 5]$ $, [6, 5, 4, 3, 2, 1, 0, 1, 2, 3, 4]$ $, [7, 6, 5, 4, 3, 2, 1, 0, 1, 2, 3]$ $, [8, 7, 6, 5, 4, 3, 2, 1, 0, 1, 2]$ $, [9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 1]$ $, [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]$])	(0,[$[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$ $, [1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$ $, [2, 1, 0, 1, 2, 3, 4, 5, 6, 7, 8]$ $, [3, 2, 1, 0, 1, 2, 3, 4, 5, 6, 7]$ $, [4, 3, 2, 1, 0, 1, 2, 3, 4, 5, 6]$ $, [5, 4, 3, 2, 1, 0, 1, 2, 3, 4, 5]$ $, [6, 5, 4, 3, 2, 1, 0, 1, 2, 3, 4]$ $, [7, 6, 5, 4, 3, 2, 1, 0, 1, 2, 3]$ $, [8, 7, 6, 5, 4, 3, 2, 1, 0, 1, 2]$ $, [9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 1]$ $, [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]$])	✓

Обычный случай	"usual"	"specul"	(5,[$[0,1,2,3,4,5]$ $,[1,1,1,2,3,4]$ $,[2,2,2,2,3,4]$ $,[3,3,3,3,3,4]$ $,[4,4,4,4,4,4]$ $,[5,4,5,4,5,5]$ $,[6,5,5,5,5,5]$])	(5,[$[0,1,2,3,4,5]$ $,[1,1,1,2,3,4]$ $,[2,2,2,2,3,4]$ $,[3,3,3,3,3,4]$ $,[4,4,4,4,4,4]$ $,[5,4,5,4,5,5]$ $,[6,5,5,5,5,5]$])	✓
Случай с совпадением 1 символа	"usual"	"qswer"	(4,[$[0,1,2,3,4,5]$ $,[1,1,2,3,4,5]$ $,[2,2,1,2,3,4]$ $,[3,3,2,2,3,4]$ $,[4,4,3,3,3,4]$ $,[5,5,4,4,4,4]$])	(4,[$[0,1,2,3,4,5]$ $,[1,1,2,3,4,5]$ $,[2,2,1,2,3,4]$ $,[3,3,2,2,3,4]$ $,[4,4,3,3,3,4]$ $,[5,5,4,4,4,4]$])	✓
Одна строка больше другой	"usual"	"more usual"	(4,[$[0,1,2,3,4,5]$ $,[1,1,2,3,4,5]$ $,[2,2,2,3,4,5]$ $,[3,3,3,3,4,5]$ $,[4,4,4,4,4,5]$ $,[5,4,5,4,5,5]$ $,[6,5,4,5,5,6]$ $,[7,6,5,4,5,6]$ $,[8,7,6,5,4,5]$ $,[9,8,7,6,5,4]$])	(4,[$[0,1,2,3,4,5]$ $,[1,1,2,3,4,5]$ $,[2,2,2,3,4,5]$ $,[3,3,3,3,4,5]$ $,[4,4,4,4,4,5]$ $,[5,4,5,4,5,5]$ $,[6,5,4,5,5,6]$ $,[7,6,5,4,5,6]$ $,[8,7,6,5,4,5]$ $,[9,8,7,6,5,4]$])	✓

Таблица 3.5: Матричный алгоритм Левенштейна с выводом матрицы.

Описание теста	Ввод		Ожидаемое	Вывод	Рез
	Строка №1	Строка №2			
Пустые строки	"_"	"_"	(0, [[0]])	(0, [[0]])	✓
Вторая строка пуста	"a"	"_"	(1, [[0, 1]])	(1, [[0, 1]])	✓
Первая строка пуста	"_"	"a"	(1, [[0], [1]])	(1, [[0], [1]])	✓
Одинаковые строки	"equal"	"equal"	(0,[[0, 1, 2, 3, 4, 5] , [1, 0, 1, 2, 3, 4] , [2, 1, 0, 1, 2, 3] , [3, 2, 1, 0, 1, 2] , [4, 3, 2, 1, 0, 1] , [5, 4, 3, 2, 1, 0]])	(0,[[0, 1, 2, 3, 4, 5] , [1, 0, 1, 2, 3, 4] , [2, 1, 0, 1, 2, 3] , [3, 2, 1, 0, 1, 2] , [4, 3, 2, 1, 0, 1] , [5, 4, 3, 2, 1, 0]])	✓
Входные строки в кириллице	"одинаковые"	"одинаковые"	(0,[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] , [1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9] , [2, 1, 0, 1, 2, 3, 4, 5, 6, 7, 8] , [3, 2, 1, 0, 1, 2, 3, 4, 5, 6, 7] , [4, 3, 2, 1, 0, 1, 2, 3, 4, 5, 6] , [5, 4, 3, 2, 1, 0, 1, 2, 3, 4, 5] , [6, 5, 4, 3, 2, 1, 0, 1, 2, 3, 4] , [7, 6, 5, 4, 3, 2, 1, 0, 1, 2, 3] , [8, 7, 6, 5, 4, 3, 2, 1, 0, 1, 2] , [9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 1] , [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]])	(0,[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] , [1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9] , [2, 1, 0, 1, 2, 3, 4, 5, 6, 7, 8] , [3, 2, 1, 0, 1, 2, 3, 4, 5, 6, 7] , [4, 3, 2, 1, 0, 1, 2, 3, 4, 5, 6] , [5, 4, 3, 2, 1, 0, 1, 2, 3, 4, 5] , [6, 5, 4, 3, 2, 1, 0, 1, 2, 3, 4] , [7, 6, 5, 4, 3, 2, 1, 0, 1, 2, 3] , [8, 7, 6, 5, 4, 3, 2, 1, 0, 1, 2] , [9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 1] , [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]])	✓

Обычный случай	"usual"	"specul"	(5,[$[0,1,2,3,4,5]$ $,[1,1,1,2,3,4]$ $,[2,2,2,2,3,4]$ $,[3,3,3,3,3,4]$ $,[4,4,4,4,4,4]$ $,[5,4,5,4,5,5]$ $,[6,5,5,5,5,5]$])	(5,[$[0,1,2,3,4,5]$ $,[1,1,1,2,3,4]$ $,[2,2,2,2,3,4]$ $,[3,3,3,3,3,4]$ $,[4,4,4,4,4,4]$ $,[5,4,5,4,5,5]$ $,[6,5,5,5,5,5]$])	✓
Случай с совпадением 1 символа	"usual"	"qswer"	(4,[$[0,1,2,3,4,5]$ $,[1,1,2,3,4,5]$ $,[2,2,1,2,3,4]$ $,[3,3,2,2,3,4]$ $,[4,4,3,3,3,4]$ $,[5,5,4,4,4,4]$])	(4,[$[0,1,2,3,4,5]$ $,[1,1,2,3,4,5]$ $,[2,2,1,2,3,4]$ $,[3,3,2,2,3,4]$ $,[4,4,3,3,3,4]$ $,[5,5,4,4,4,4]$])	✓
Одна строка больше другой	"usual"	"more usual"	(4,[$[0,1,2,3,4,5]$ $,[1,1,2,3,4,5]$ $,[2,2,2,3,4,5]$ $,[3,3,3,3,4,5]$ $,[4,4,4,4,4,5]$ $,[5,4,5,4,5,5]$ $,[6,5,4,5,5,6]$ $,[7,6,5,4,5,6]$ $,[8,7,6,5,4,5]$ $,[9,8,7,6,5,4]$])	(4,[$[0,1,2,3,4,5]$ $,[1,1,2,3,4,5]$ $,[2,2,2,3,4,5]$ $,[3,3,3,3,4,5]$ $,[4,4,4,4,4,5]$ $,[5,4,5,4,5,5]$ $,[6,5,4,5,5,6]$ $,[7,6,5,4,5,6]$ $,[8,7,6,5,4,5]$ $,[9,8,7,6,5,4]$])	✓

Случай с перестановкой двух соседних символов	"mroe suua"	"more usual"	(3,[[0, 1, 2, 3, 4, 5, 6, 7, 8] , [1, 0, 1, 2, 3, 4, 5, 6, 7] , [2, 1, 1, 1, 2, 3, 4, 5, 6] , [3, 2, 1, 1, 2, 3, 4, 5, 6] , [4, 3, 2, 2, 1, 2, 3, 4, 5] , [5, 4, 3, 3, 2, 2, 2, 3, 4] , [6, 5, 4, 4, 3, 2, 2, 3, 4] , [7, 6, 5, 5, 4, 3, 2, 2, 3] , [8, 7, 6, 6, 5, 4, 3, 3, 2] , [9, 8, 7, 7, 6, 5, 4, 4, 3]])	(3,[[0, 1, 2, 3, 4, 5, 6, 7, 8] , [1, 0, 1, 2, 3, 4, 5, 6, 7] , [2, 1, 1, 1, 2, 3, 4, 5, 6] , [3, 2, 1, 1, 2, 3, 4, 5, 6] , [4, 3, 2, 2, 1, 2, 3, 4, 5] , [5, 4, 3, 3, 2, 2, 2, 3, 4] , [6, 5, 4, 4, 3, 2, 2, 3, 4] , [7, 6, 5, 5, 4, 3, 2, 2, 3] , [8, 7, 6, 6, 5, 4, 3, 3, 2] , [9, 8, 7, 7, 6, 5, 4, 4, 3]])	✓

3.5 Выводы

- Был выбран язык программирования Haskell для реализации поставленной задачи;
- Был найден способ (функция) замера процессорного времени на языке Haskell;
- Были написаны тесты к программе;
- Была написана программа, удовлетворяющая всем требованиям, поставленным в конструкторской части;

4 Исследовательская часть

4.1 Замер времени

Был проведен замер времени работы каждого из алгоритмов.

Таблица 4.1: Время работы алгоритмов.

len	DamLev(R), нс	DamLev(T), нс	Lev(T), нс
3	0.05122	0.00666	0.0118
4	0.21172	0.01661	0.01653
5	0.82129	0.01867	0.01785
6	3.90051	0.0322	0.02913
7	21.88572	0.04306	0.03041
8	131.24082	0.04461	0.04229
9	789.49599	0.05559	0.04601

Где len - длина слова (подразумевается что длины обоих слов одинаковы, это было сделано для удобства) , DamLev(R) - Время работы рекурсивного алгоритма Дамерау-Левенштейна, Lev(T) - Время работы матричного алгоритма Левенштейна, DamLev(T) - Время работы матричного алгоритма Дамерау-Левенштейна.

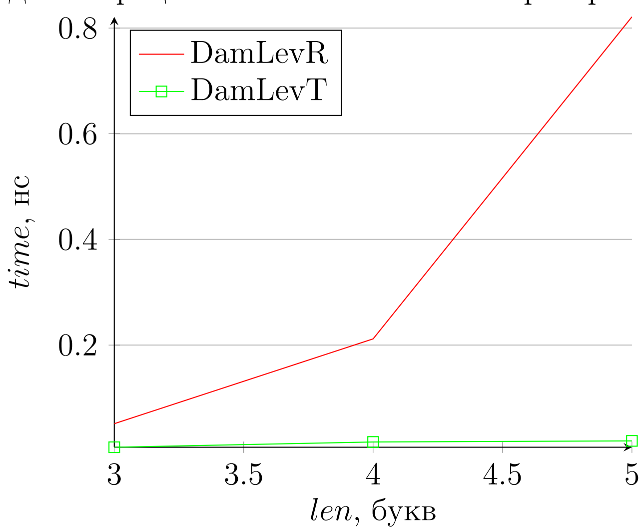
Время работы представлено в наносекундах. Для получения более точного результата процессорное время считалось как сумма всех времен, потраченных на каждый эксперимент, деленная на их количество:

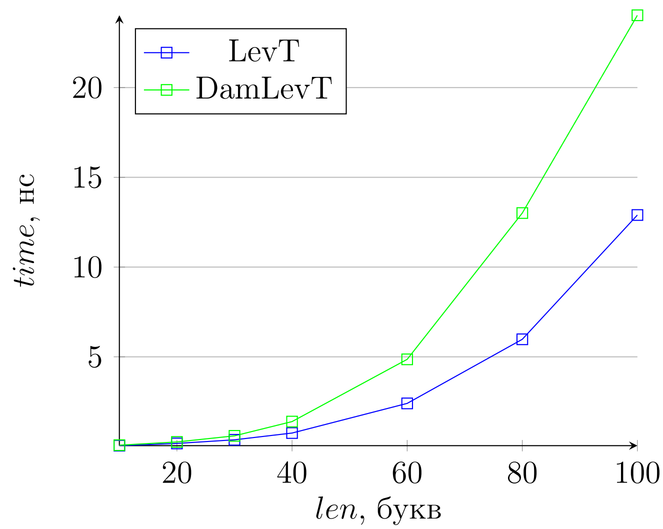
$$T = \frac{\sum T_i}{N}, i \in [1, N]$$

Таблица 4.2: Время работы алгоритмов.

len	DamLev(T), нс	Lev(T), нс
10	0.07763	0.05143
20	0.25776	0.17495
30	0.5983	0.37773
40	1.39289	0.75407
60	4.85997	2.40581
80	13.00983	5.97663
100	24.0279	12.90033

Где количество измерений было решено сделать равным 100. Строки на каждой итерации были составлены генератором случаных символов.





4.2 Выводы

Исходя из представленных графиков, можно сделать следующие выводы:

- матричная реализация алгоритмов Дамерау-Левенштейна **значительно** быстрее рекурсивной. Уже при длине слова равном 5 результат различается почти в 80 раз, и далее растет экспоненциально;

- на больших входных данных (длины слов равны 100) матричная реализация алгоритма Дамерау-Левенштейна начинает быть почти в 2 раза менее эффективной чем матричная реализация алгоритма Левенштейна.

Таким образом, рекурсивная реализация (как и ожидалось), оказалась абсолютно бесполезным решением для данной задачи. Матричная реализация эффективнее как по памяти, так и по времени на **любых** (не считая пустых значений) входных данных.

А матричная реализация алгоритма Дамерау-Левенштейна справляется с поставленной задачей значительно медленнее (почти в 2 раза медленнее) чем реализация алгоритма Левенштейна на больших входных данных. (длины слов по 100 букв)

Заключение

В рамках данной работы были изучены различные **алгоритмы нечеткого поиска**. А именно:

- Алгоритм Левенштейна;
- Алгоритм Дамерау-Левенштейна;

А также их реализации:

- Рекурсивный;
- Матричный;

Экспериментально было подтверждено, что рекурсивная реализация гораздо медленнее эффективна как по времени, так и по памяти. Уже при значениях входных данных больше 5, разница во времени составляет больше чем в 80 раз. Это обусловливается тем, что рекурсивная реализация на каждом этапе своей рекурсии разбивает задачу на более мелкие, при этом не контролируя возможность образования повторяющихся случаев.

В то время как матричная реализация использует обычный итеративный подход, не требующий больших затрат по памяти. (для вычисления следующей строки нам достаточно запомнить только 1-2 предыдущие строки)

Также, алгоритм Левенштейна оказался более эффективным как по времени, так и по памяти, нежели алгоритм Дамерау-Левенштейна. При проведении эксперимента над их матричными реализациями оказалось, что время, затраченное на выполнение поставленной задачи, на больших входных данных (порядка 100 букв в слове) различается почти в 2 раза. Что неудивительно, ведь алгоритм Дамерау-Левенштейна добавляет дополнительный функционал к алгоритму Левенштейна, который проверяет входное слово на наличие опечаток. (две соседние буквы поменялись местами, подробнее см. Аналит. часть)

Литература

[1] https://www.haskell.org/documentation/ - Документация Haskell

[2] http://repo.ssau.ru/bitstream/Informacionnye-tehnologii-i-nanotehnologii/Algoritm-nechetkogo-poiska-v-bazah-dannyh-i-ego-prakticheskaya-realizaciya-64172/1/paper - Алгоритмы нечеткого поиска

[3] Двоичные коды с исправлением выпадений, вставок и замещений символов. Доклады Академий Наук СССР, 1965. В. И. Левенштейн.

[4] A technique for computer detection and correction of spelling errors. Damerau Fred J.

[5] Indexing methods for approximate dictionary searching. Journal of Experimental Algorithmics, 2011. L. M. Boytsov