



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 Расстояния Левенштейна и Дамерау-Левенштейна

Студент Лемешев А. П.

Группа ИУ7-52Б

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

Оглавление

Введение	2
1 Аналитическая часть	3
1.1 Расстояние Левенштейна	3
1.2 Расстояние Дамерау-Левенштейна	4
1.3 Итеративная реализация	5
1.4 Рекурсивная реализация с мемоизацией	6
2 Конструкторская часть	7
2.1 Алгоритм Левенштейна	7
2.2 Алгоритмы Дамерау-Левенштейна	8
3 Технологическая часть	11
3.1 Требования к ПО	11
3.2 Средства реализации	11
3.3 Листинг кода	11
3.4 Тестирование функций	20
4 Исследовательская часть	21
4.1 Технические характеристики	21
4.2 Пример работы программы	21
4.3 Время выполнения алгоритмов	22
4.4 Использование памяти	23
4.4.1 Нерекурсивные алгоритмы	23
4.4.2 Рекурсивные алгоритмы	24
Заключение	25
Список использованных источников	26

Введение

Нахождение редакционного расстояния – одна из задач компьютерной лингвистики, которая находит применение в огромном количестве областей, например:

- исправление ошибок в тексте в поисковых запросах;
- сравнение текстовых файлов утилитой diff [1];
- сравнение генов, хромосом и белков в биоинформатике.

Впервые задачу поставил советский ученый В. И. Левенштейн при изучении последовательностей 0-1 [2]. Впоследствии более общую задачу для произвольного алфавита связали с его именем. Позже Фредерик Дамерау заявил, что при исследовании орфографических ошибок в информационно-поисковых системах более 80% человеческих ошибок при наборе текстов составляют перестановки соседних символов, пропуск символа, добавление нового символа и ошибка в символе.

Алгоритмы имеют некоторое количество модификаций, позволяющих эффективнее решать поставленную задачу.

1 Аналитическая часть

В этом разделе будут представлены описания алгоритмов нахождения редакторских расстояний Левенштейна и Дameraу-Левенштейна.

Цель лабораторной работы – разработка, реализация и исследование алгоритмов нахождения расстояний Левенштейна и Дameraу-Левенштейна.

Задачи лабораторной работы:

- изучение алгоритмов редакционных расстояний Левенштейна и Дameraу-Левенштейна;
- получение практических навыков реализаций алгоритмов редакционных расстояний Левенштейна и Дameraу-Левенштейна;
- проведение сравнительного анализа алгоритмов определения расстояния между строками по затратам времени и памяти;
- применение метода динамического программирования для реализации алгоритмов;
- описание и обоснование полученных результатов в отчете о выполненной лабораторной работе.

1.1 Расстояние Левенштейна

Расстояние Левенштейна – метрика, измеряющая разность двух строк символов, определяемая в количестве редакторских операций (а именно удаления, вставки и замены), требуемых для преобразования одной последовательности в другую. Каждая редакторская операция имеет цену (штраф). Редакционное предписание – последовательность действий, необходимых для получения из первой строки вторую, и минимизирующую суммарную цену (и является расстоянием Левенштейна). В общем случае, имея на входе строку, $X = x_1x_2 \dots x_n$, и, $Y = y_1y_2 \dots y_n$, расстояние между ними можно вычислить с помощью операций:

- $\text{delete}(u, \varepsilon) = \delta$;

- $\text{insert}(\varepsilon, v) = \delta$;
- $\text{replace}(u, v) = \alpha(u, v) \leq 0$ (здесь, $\alpha(u, u) = 0 \forall u$).

Необходимо найти последовательность замен с минимальным суммарным штрафом. Далее, цена вставки, удаления и замены будет считаться равной 1. Пусть даны строки: $s1 = s1[1..L1]$, $s2 = s2[1..L2]$, $s1[1..i]$ – подстрока $s1$ длиной i , начиная с 1-го символа, $s2[1..j]$ – подстрока $s2$ длиной j , начиная с 1-го символа. Расстояние Левенштейна посчитывается формулой 1.1:

$$D(S_1[1..i], S_2[1..j]) = \begin{cases} 0, i = 0, j = 0, \\ i, i > 0, j = 0, \\ j, i = 0, j > 0, \\ \min\{ \\ \quad D(S_1[1..i], S_2[1..j-1]) + 1, \\ \quad D(S_1[1..i-1], S_2[1..j]) + 1, \\ \quad D(S_1[1..i-1], S_2[1..j-1]) + \\ \quad + \begin{cases} 0, \text{если } S_1[i] = S_2[j] \\ 1, \text{иначе} \end{cases} \\ \}, \text{иначе,} \end{cases} \quad (1.1)$$

где i – длина S_1 , j – длина S_2 .

1.2 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна – модификация, добавляющая к редакторским операциям транспозицию (обмен двух соседних символов местами).

Используя условные обозначения, рекурсивная формула для нахождения расстояния Дамерау-Левенштейна, $f(i, j)$, между подстроками, $x_1 \dots x_i$ и, $y_1 \dots y_j$, имеет следующий вид 1.2:

$$D(S_1[1..i], S_2[1..j]) = \begin{cases} 0, i = 0, j = 0, \\ i, i > 0, j = 0, \\ j, i = 0, j > 0, \\ \min\{ \\ \quad D(S_1[1..i], S_2[1..j - 1]) + 1, \\ \quad D(S_1[1..i - 1], S_2[1..j]) + 1, \\ \quad D(S_1[1..i - 1], S_2[1..j - 1]) + \\ \quad + \begin{cases} 0, \text{ если } S_1[i] = S_2[j] \\ 1, \text{ иначе} \end{cases} \\ \quad D(S_1[1..i - 2], S_2[1..j - 2]) + \\ \quad + f(S_1, i, S_2, j) \\ \}, \text{ иначе,} \end{cases} \quad (1.2)$$

где i – длина S_1 , j – длина S_2 и где функция f определяется формулой 1.3:

$$f(S_1, i, S_2, j) = \begin{cases} 1, S_1[i - 1] = S_2[j] \text{ и } S_1[i] = S_2[j - 1] \\ 2, \text{ иначе} \end{cases} \quad (1.3)$$

1.3 Итеративная реализация

Рекурсивный алгоритм вычисления редакционного расстояния может быть не эффективен при больших i и j , так как множество промежуточных значений $D(S_1[1..i], S_2[1..j])$ вычисляются не один раз, что сильно замедляет время выполнения программы. В качестве оптимизации можно использовать буфер, который представляет из себя массив, для хранения промежуточных значений. Буфер имеет размер 1.4:

$$\text{len}(S_1) + 1 \quad (1.4)$$

где $\text{len}(S_1)$ – длина строки S_1 .

Первые значения буфера заполняются по формуле 1.5:

$$buf[i] = i \quad (1.5)$$

где i – i -ый элемент массива.

Далее значения буфера меняются по формуле 1.6:

$$buf[i] = \min\{buf[j-1] + 1, buf[j] + 1, buf[j] + \begin{cases} 0, \text{ если } S_1[i] = S_2[j] \\ 1, \text{ иначе} \end{cases}\} \quad (1.6)$$

где i – длина S_1 , j – длина S_2 .

Результатом вычисления расстояния Дамерау-Левенштейна будет ячейка буфера с индексом $i = length(S_1)$.

1.4 Рекурсивная реализация с мемоизацией

В качестве оптимизации рекурсивного алгоритма заполнения можно использовать кэш, который будет представлять собой матрицу. Суть оптимизации – при выполнении рекурсии происходит параллельное заполнение матрицы. Если рекурсивный алгоритм выполняет прогон для данных, которые еще не были обработаны, то результат нахождения заносится в матрицу. Иначе, если обработанные данные встречаются снова, то для них расстояние не находится и алгоритм переходит к следующему шагу.

Вывод

Были рассмотрены обе вариации алгоритма редакторского расстояния (Левенштейна и Дамерау-Левенштейна). Также были приведены разные способы реализации этих алгоритмов такие как: рекурсивный, итеративный и рекурсивный с мемоизацией. Итеративный может быть реализован с помощью парадигм динамического программирования. Рекурсивный алгоритм с мемоизацией позволяет ускорить обычный рекурсивный алгоритм за счет буфера, в котором содержатся промежуточные подсчеты.

2 Конструкторская часть

В данном разделе представлены схемы реализуемых алгоритмов.

2.1 Алгоритм Левенштейна

На рисунке 2.1 представлена схема алгоритма нерекурсивного метода поиска расстояния Левенштейна.

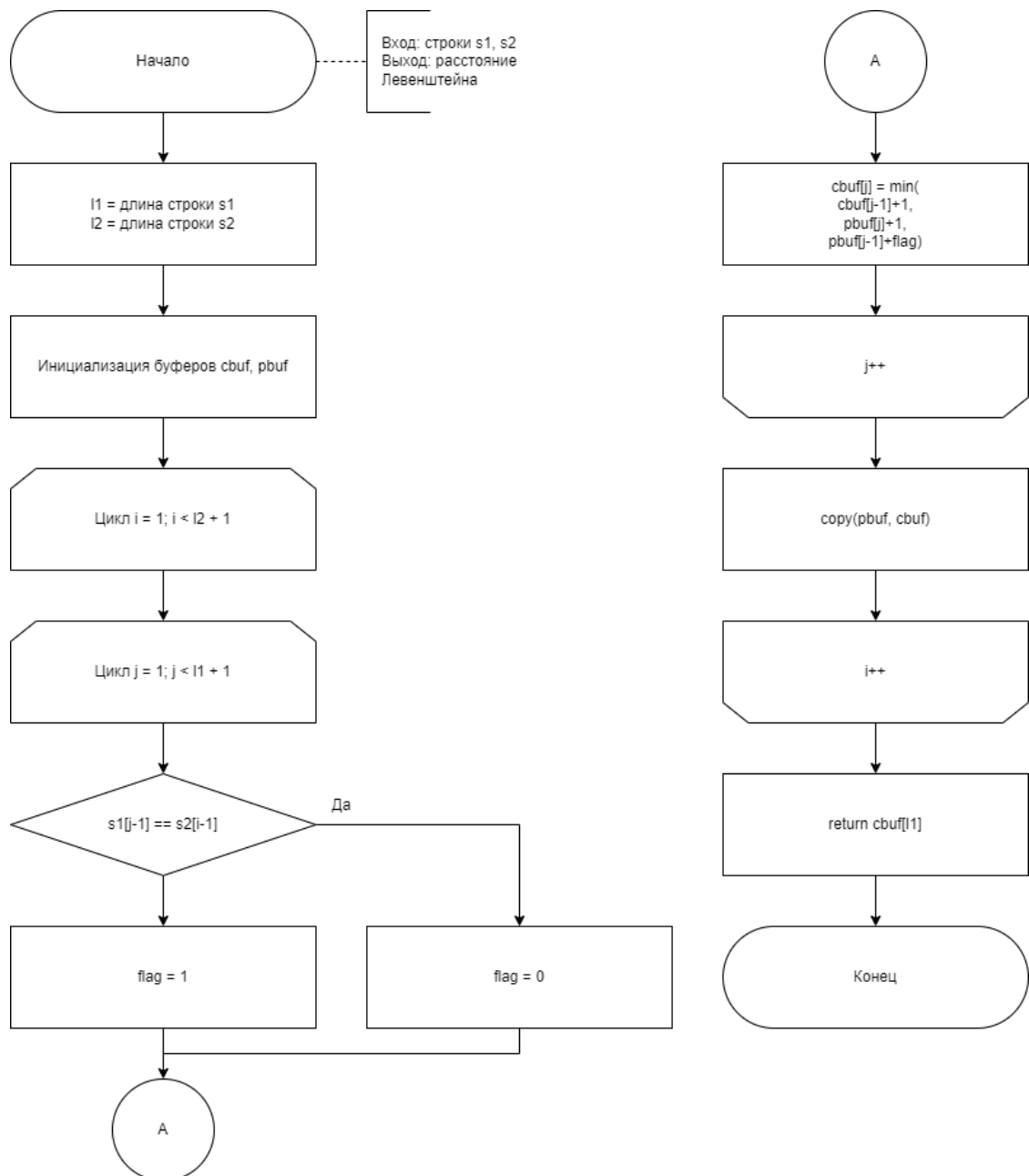


Рис. 2.1: Схема нерекурсивного алгоритма Левенштейна

2.2 Алгоритмы Дамерау-Левенштейна

На рисунках 2.2, 2.3 и 2.4 представлены схемы алгоритмов нерекурсивного, рекурсивного и рекурсивного с кешированием методов поиска расстояния Дамерау-Левенштейна.



Рис. 2.2: Схема нерекурсивного алгоритма Дамерау-Левенштейна

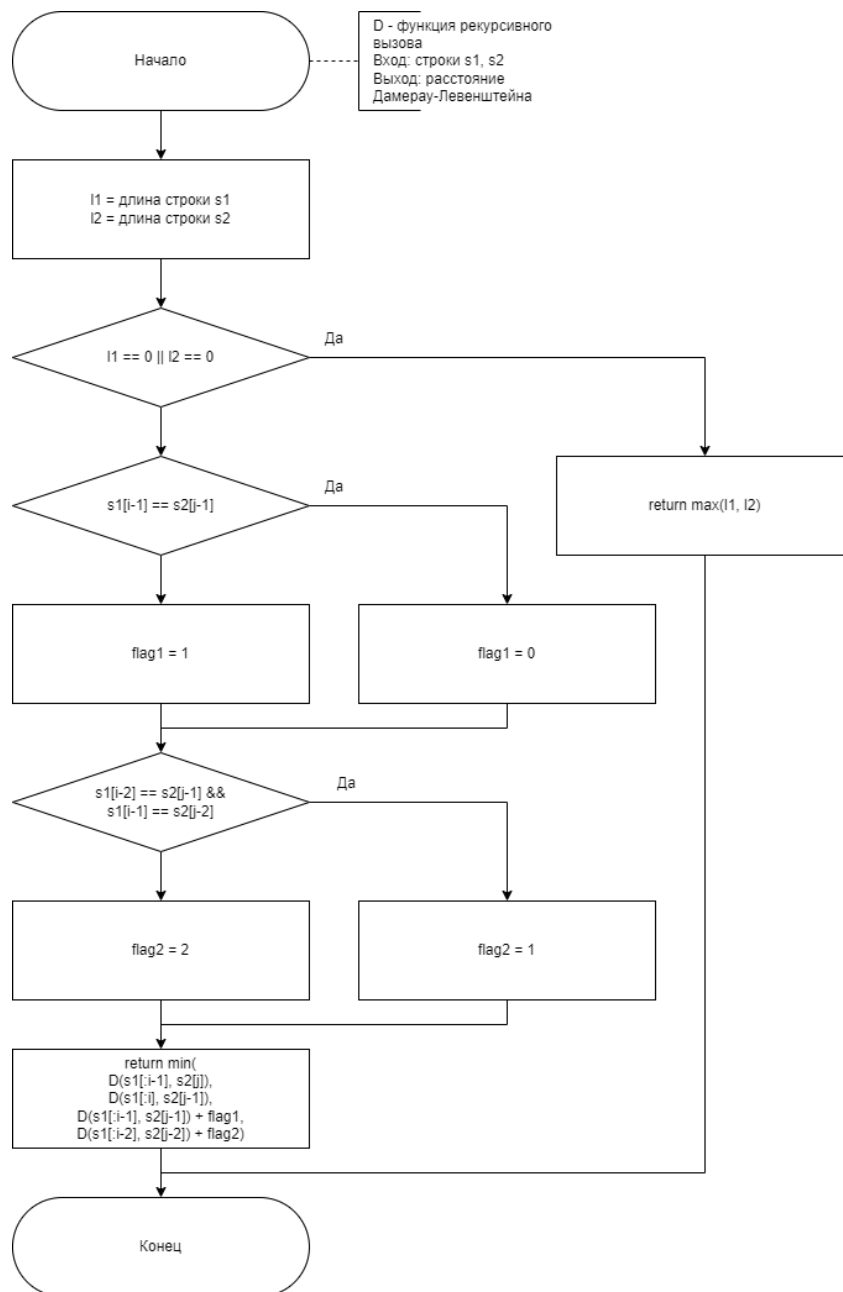


Рис. 2.3: Схема рекурсивного алгоритма Дameraу-Левенштейна

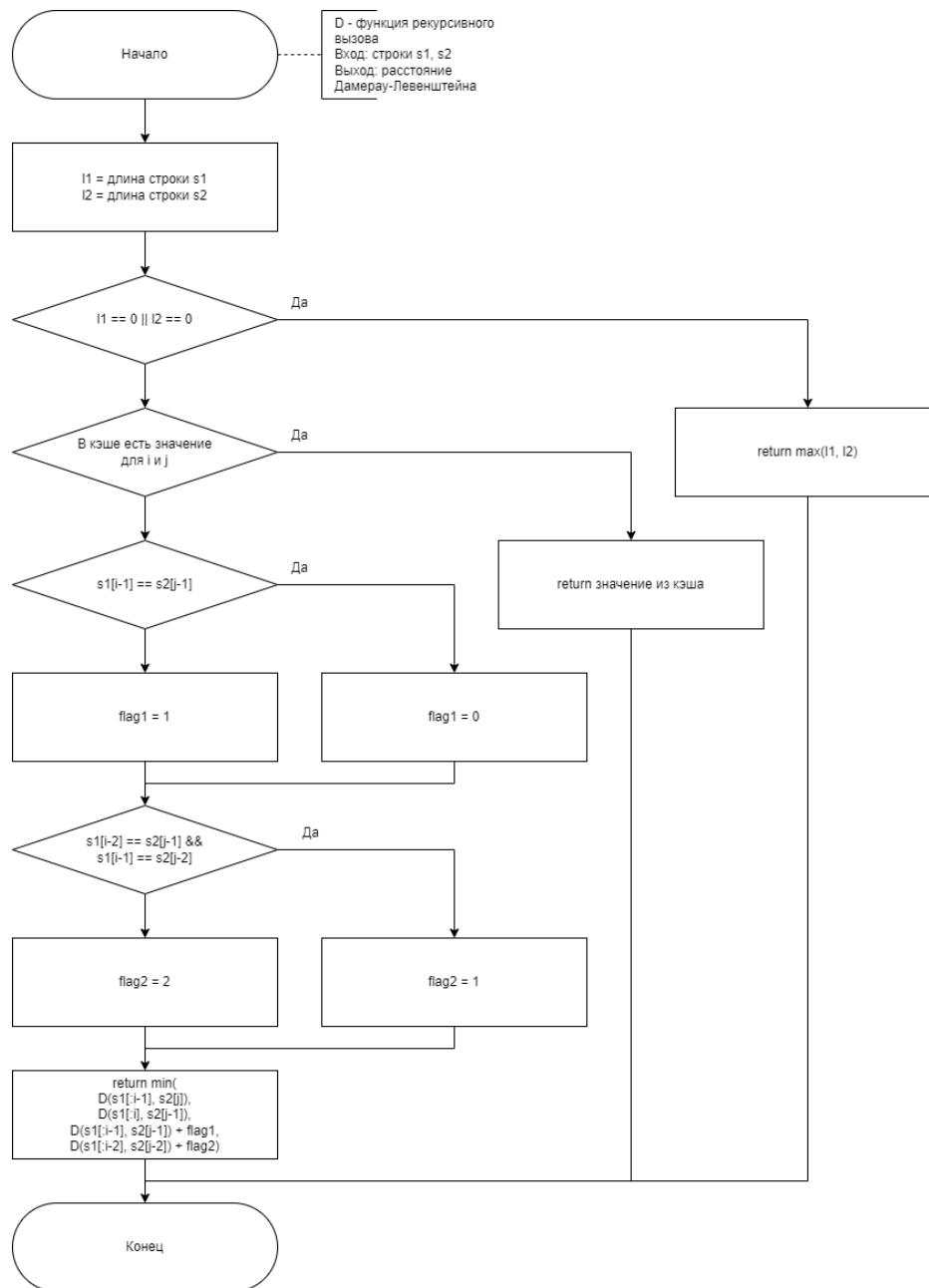


Рис. 2.4: Схема рекурсивного алгоритма с кешированием Дамерау-Левенштейна

Вывод

На основе формул и теоретических данных, полученных в аналитическом разделе, были спроектированы схемы алгоритмов.

3 Технологическая часть

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

3.1 Требования к ПО

Программа должна отвечать следующим требованиям:

- программа на вход принимает две строки;
- программа выдает редакционное расстояние для всех четырех методов.

3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран современный компилируемый ЯП Golang [3]. Данный выбор обусловлен моим желанием расширить свои знания в области применения данного языка, а также тем, что данный язык предоставляет широкие возможности для написания тестов [4].

3.3 Листинг кода

В листингах 3.1 – 3.4 приведены листинги описанных алгоритмов Левенштейна и Дамерау-Левенштейна. В листинге 3.5 приведены вспомогательные функции. В листингах 3.6 и 3.8 приведены примеры реализации тестов и бенчмарков.

Листинг 3.1: Итеративный алгоритм Левенштейна

```
1 package levenshtein
2
3 import "lab_01/utils"
4
5 func (l Levenshtein) LIterative() int {
6     l1, l2 := l.lens()
7     if l.isEmpty() {
8         return utils.MaxFromSome(l1, l2)
9     }
10    cbuf := make([]int, l1+1) // cur buffer
11    pbuf := make([]int, l1+1) // prev buffer
12    for i := range pbuf {
13        pbuf[i] = i
14    }
15    for i := 1; i < l2+1; i++ {
16        cbuf[0] = i
17        for j := 1; j < l1+1; j++ {
18            eq := l.isEqualRunes(j-1, i-1)
19            res := utils.MinFromSome(
20                cbuf[j-1]+1,
21                pbuf[j]+1,
22                pbuf[j-1]+eq,
23            )
24            cbuf[j] = res
25        }
26        copy(pbuf, cbuf)
27    }
28    return cbuf[l1]
29 }
```

Листинг 3.2: Итеративный алгоритм Дамерау-Левенштейна

```
1 package levenshtein
2
3 import "lab_01/utils"
4
5 func (l Levenshtein) DLIterative() int {
6     l1, l2 := l.lens()
7     if l.isEmpty() {
8         return utils.MaxFromSome(l1, l2)
9     }
10    cbuf := make([]int, l1+1) // cur buffer
11    p1buf := make([]int, l1+1) // prev1 buffer
12    p2buf := make([]int, l1+1) // prev2 buffer
13    for i := range p1buf {
14        p1buf[i] = i
15    }
16    for i := 1; i < l2+1; i++ {
17        cbuf[0] = i
18        for j := 1; j < l1+1; j++ {
19            eq := l.isEqualRunes(j-1, i-1)
20            res := utils.MinFromSome(
21                cbuf[j-1]+1,
22                p1buf[j]+1,
23                p1buf[j-1]+eq,
24            )
25            if l.dlFlag(j, i) {
26                res = utils.MinFromSome(
27                    res,
28                    p2buf[j-2]+1,
29                )
30            }
31            cbuf[j] = res
32        }
33        copy(p2buf, p1buf)
34        copy(p1buf, cbuf)
35    }
36    return cbuf[l1]
37 }
```

Листинг 3.3: Рекурсивный алгоритм Дамерау-Левенштейна

```
1 package levenshtein
2
3 import "lab_01/utils"
4
5 func (l Levenshtein) DLRecursive() int {
6     l1, l2 := l.lens()
7     if l.isEmpty() {
8         return utils.MaxFromSome(l1, l2)
9     }
10    eq := l.isEqualRunes(l1-1, l2-1)
11    res := utils.MinFromSome(
12        l.cutRune(1, 0).DLRecursive()+1,
13        l.cutRune(0, 1).DLRecursive()+1,
14        l.cutRune(1, 1).DLRecursive()+eq,
15    )
16    if l.dlFlag(l1, l2) {
17        res = utils.MinFromSome(
18            res,
19            l.cutRune(2, 2).DLRecursive()+1,
20        )
21    }
22    return res
23 }
```

Листинг 3.4: Рекурсивный алгоритм с кэшем Дамерау-Левенштейна

```
1 package levenshtein
2
3 import "lab_01/utils"
4
5 func (l Levenshtein) DLRecursiveCash() int {
6     l1, l2 := l.lens()
7     if l.isEmpty() {
8         return utils.MaxFromSome(l1, l2)
9     }
10    c := make([] []int, l1) // cache
11    for i := range c {
12        c[i] = make([]int, l2)
13        for j := range c[i] {
14            c[i][j] = -1
15        }
16    }
17    return l.dlRecursiveCash(c)
18 }
19
20 func (l Levenshtein) dlRecursiveCash(c [] []int) int {
21     l1, l2 := l.lens()
22     if l.isEmpty() {
23         return utils.MaxFromSome(l1, l2)
24     }
25     if c[l1-1][l2-1] != -1 {
26         return c[l1-1][l2-1]
27     }
28     eq := l.isEqualRunes(l1-1, l2-1)
29     res := utils.MinFromSome(
30         l.cutRune(1, 0).dlRecursiveCash(c)+1,
31         l.cutRune(0, 1).dlRecursiveCash(c)+1,
32         l.cutRune(1, 1).dlRecursiveCash(c)+eq,
33     )
34     if l.dlFlag(l1, l2) {
35         res = utils.MinFromSome(
36             res,
37             l.cutRune(2, 2).DLRecursive()+1,
38         )
39     }
40     c[l1-1][l2-1] = res
41     return res
42 }
```


Листинг 3.5: Вспомогательные функции для расчёта расстояний

```
1 package levenshtein
2
3 type Levenshtein struct {
4     runes1 []rune
5     runes2 []rune
6 }
7
8 func Make(str1, str2 string) Levenshtein {
9     return Levenshtein{[]rune(str1), []rune(str2)}
10 }
11
12 func (l Levenshtein) cutRune(c1, c2 int) Levenshtein {
13     l1, l2 := l.lens()
14     return Levenshtein{l.runes1[:l1-c1], l.runes2[:l2-c2]}
15 }
16
17 func (l Levenshtein) lens() (int, int) {
18     return len(l.runes1), len(l.runes2)
19 }
20
21 func (l Levenshtein) isEmpty() bool {
22     return len(l.runes1) == 0 || len(l.runes2) == 0
23 }
24
25 func (l Levenshtein) isEqualRunes(i, j int) int {
26     if l.runes1[i] == l.runes2[j] {
27         return 0
28     }
29     return 1
30 }
31
32 func (l Levenshtein) dlFlag(i, j int) bool {
33     if i <= 1 {
34         return false
35     } else if j <= 1 {
36         return false
37     } else if l.runes1[i-1] != l.runes2[j-2] {
38         return false
39     } else if l.runes1[i-2] != l.runes2[j-1] {
40         return false
41     }
42     return true
43 }
```

Листинг 3.6: Пример реализации тестов

```

1 package levenshtein
2
3 import "testing"
4
5 var testLevenshteinTable = []struct {
6     title string
7     in []string
8     lOut int
9     dlOut int
10 }{
11     {
12         title: "equal_strings",
13         in: []string{"aaasssddd", "aaasssddd"},
14         lOut: 0,
15         dlOut: 0,
16     },
17     {
18         title: "strl_smaller",
19         in: []string{"aaasssdd", "aaasssddd"},
20         lOut: 1,
21         dlOut: 1,
22     },
23     // ...
24 }
25
26 func TestLIterative(t *testing.T) {
27     for _, test := range testLevenshteinTable {
28         l := Make(test.in[0], test.in[1])
29         res := l.LIterative()
30         if test.lOut != res {
31             t.Errorf("Incorrect_result.\ntitle:%v\nin:%v\nout:%v\nres:\n%v\n",
32                 test.title, test.in, test.lOut, res)
33         } else {
34             t.Logf("Test_pass_%v'.\n", test.title)
35         }
36     }
37 }

```

Листинг 3.7: Пример реализации тестов

```

1 func TestDLIterative(t *testing.T) {
2     for _, test := range testLevenshteinTable {
3         l := Make(test.in[0], test.in[1])
4         res := l.DLIterative()
5         if test.dlOut != res {
6             t.Errorf("Incorrect result.\ntitle:_%v\nin:_%v\nout:_%v\nres:_%v\n",
7                 test.title, test.in, test.dlOut, res)
8         } else {
9             t.Logf("Test pass_%v'\n", test.title)
10        }
11    }
12 }
13
14 func TestDLRecursive(t *testing.T) {
15     for _, test := range testLevenshteinTable {
16         l := Make(test.in[0], test.in[1])
17         res := l.DLRecursive()
18         if test.dlOut != res {
19             t.Errorf("Incorrect result.\ntitle:_%v\nin:_%v\nout:_%v\nres:_%v\n",
20                 test.title, test.in, test.dlOut, res)
21         } else {
22             t.Logf("Test pass_%v'\n", test.title)
23         }
24     }
25 }
26
27 func TestDLRecursiveCash(t *testing.T) {
28     for _, test := range testLevenshteinTable {
29         l := Make(test.in[0], test.in[1])
30         res := l.DLRecursiveCash()
31         if test.dlOut != res {
32             t.Errorf("Incorrect result.\ntitle:_%v\nin:_%v\nout:_%v\nres:_%v\n",
33                 test.title, test.in, test.dlOut, res)
34         } else {
35             t.Logf("Test pass_%v'\n", test.title)
36         }
37     }
38 }

```

Листинг 3.8: Пример реализации бенчмарка

```
1 package levenshtein
2
3 import "testing"
4
5 var data = [3][2]string{
6     {
7         "dog",
8         "god",
9     },
10 }
11
12 func BenchmarkLIterativeLen3(b *testing.B) {
13     l := Make(data[0][0], data[0][1])
14     for i := 0; i < b.N; i++ {
15         l.LIterative()
16     }
17 }
18
19 func BenchmarkDLIterativeLen3(b *testing.B) {
20     l := Make(data[0][0], data[0][1])
21     for i := 0; i < b.N; i++ {
22         l.DLIterative()
23     }
24 }
25
26 func BenchmarkDLRecursiveLen3(b *testing.B) {
27     l := Make(data[0][0], data[0][1])
28     for i := 0; i < b.N; i++ {
29         l.DLRecursive()
30     }
31 }
32
33 func BenchmarkDLRecursiveCashLen3(b *testing.B) {
34     l := Make(data[0][0], data[0][1])
35     for i := 0; i < b.N; i++ {
36         l.DLRecursiveCash()
37     }
38 }
```

3.4 Тестирование функций

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

Таблица 3.1: Тестовые данные

Строка 1	Строка 2	Левенштейн	Дамерау-Левенштейн
cook	cooker	2	2
aboba	aboba	0	0
абвгдеё	абвг	3	3
	qwer	4	4
qwerty	ytrewq	6	5
qwerty	wqreyt	4	3

Вывод

На основе схем из конструкторского раздела были разработаны и протестированы спроектированные алгоритмы.

4 Исследовательская часть

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

4.1 Технические характеристики

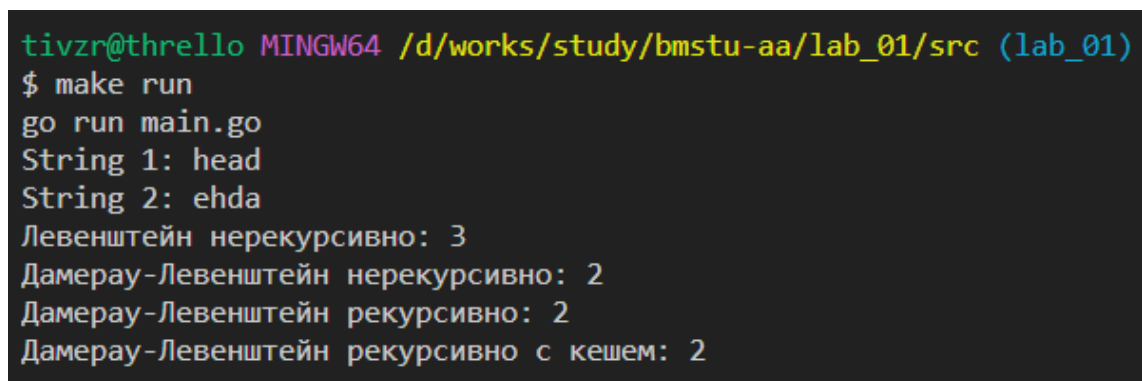
Тестирование выполнялось на устройстве со следующими техническими характеристиками:

- Операционная система: Windows 11 x64 [5].
- Память: 8 GiB.
- Процессор: AMD Ryzen 5 3550H [6].

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

4.2 Пример работы программы

На рисунке 4.1 представлен результат работы программы.



```
tivzr@thrello MINGW64 /d/works/study/bmstu-aa/lab_01/src (lab_01)
$ make run
go run main.go
String 1: head
String 2: ehda
Левенштейн нерекурсивно: 3
Дамерау-Левенштейн нерекурсивно: 2
Дамерау-Левенштейн рекурсивно: 2
Дамерау-Левенштейн рекурсивно с кешем: 2
```

Рис. 4.1: Пример работы программы

4.3 Время выполнения алгоритмов

Результаты тестирования приведены в таблице 4.1. Прочерк в таблице означает, что тестирование для этого набора данных не выполнялось. На рисунках 4.2 и 4.3 приведены зависимости времени работы алгоритма от длины строк.

Таблица 4.1: Время работы алгоритмов

Размер	Время работы, нс			
	Л.Итер.	Д.Л.Итер.	Д.Л.Рек.	Д.Л.Рек.кэш
5	409	584	66583	2821
10	1159	1746	359403375	12963
20	4058	6272	-	43204
50	26833	41885	-	250425
100	100431	153553	-	1051785
200	412655	657659	-	4343872

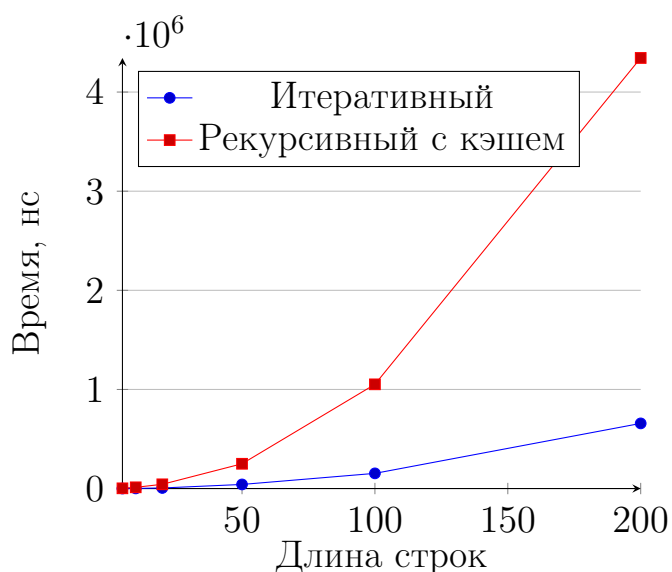


Рис. 4.2: Зависимость времени работы алгоритма вычисления расстояния Дамерау-Левенштейна от длины строк (итеративный и рекурсивный с кэшем)

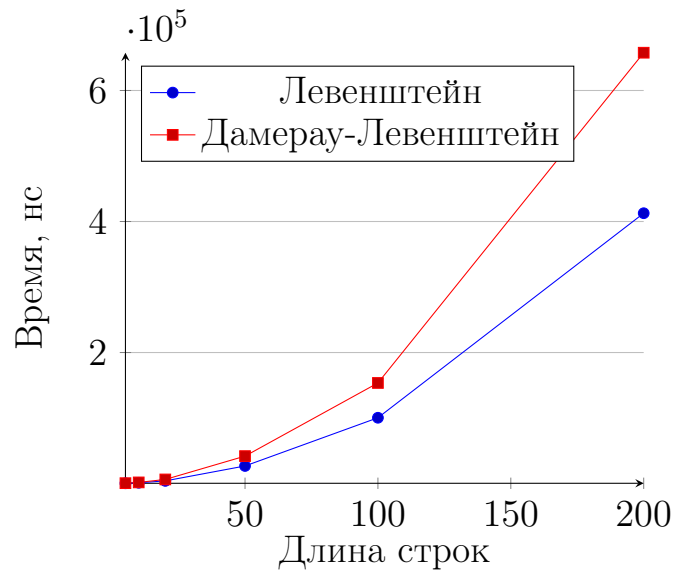


Рис. 4.3: Зависимость времени работы алгоритма вычисления расстояния Левенштейна и Дамерау-Левенштейна от длины строк (оба итеративные)

4.4 Использование памяти

Далее будем считать, что $C()$ – оператор вычисления размера. А n и m – длины строк S_1 и S_2 соответственно.

4.4.1 Нерекурсивные алгоритмы

Размер выделяемой памяти:

- $n + m$ – входные строки;
- $2 \cdot C(int)$ – длины строк;
- $2 \cdot C(slice)$ – буферы значений;
- $2 \cdot C(int)$ – вспомогательные переменные в циклах.

Общая затраченная память равняется: $n + m + 4 \cdot C(int) + 2 \cdot C(slice)$.

В алгоритме Дамерау-Левенштейна добавляется еще один буфер, для проверки перестановки. Поэтому размер выделяемой памяти, относительно

простого алгоритма Левенштейна, изменится только в вычисляемой памяти для буферов: $3 \cdot C(slice)$.

Итоговое количество затраченной памяти равняется: $n + m + 4 \cdot C(int) + 3 \cdot C(slice)$.

4.4.2 Рекурсивные алгоритмы

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящий строк $n + m$.

Для каждого вызова функции выделяемая память:

- $n + m$ – входные строки;
- $4 \cdot C(int)$ – вспомогательные переменные.

Общая затраченная память равняется: $(n + m) \cdot (n + m + 4 \cdot C(int))$.

В рекурсивном алгоритме с кэшем дополнительно выделяется память под матрицу значений: $n \cdot m \cdot C(int)$, а так же в каждую функцию передается кэш – $(\square\square int)$.

Итоговое количество затраченной памяти равняется: $(n + m) \cdot (n + m + 4 \cdot C(int) + C([n][m]int)) + n + m$.

Вывод

В данном разделе были сравнены реализованные алгоритмы по памяти и по времени. Рекурсивный алгоритм Дамерау-Левенштейна работает на порядок дольше, чем итеративная реализация этого же алгоритма. Время работы рекурсивного алгоритма увеличивается в геометрической прогрессии с ростом размера строк. Алгоритм Левенштейна работает быстрее и затрачивает меньше памяти, чем модификация Дамерау. По расходу памяти лучше всего показал себя рекурсивный алгоритм, так как для итеративных алгоритмов выделяется память под буферы, а для рекурсивного алгоритма с кэшированием необходима дополнительная матрица.

Заключение

В рамках лабораторной работы были:

- изучены алгоритмы редакционных расстояний Левенштейна и Дамерау-Левенштейна;
- получены практические навыки реализации данных алгоритмов;
- получены навыки динамического программирования;
- проведены анализы затрат работы программы по времени и по памяти.

По итогу реализации алгоритмов поиска редакционного расстояния итеративный способ оказался быстрее рекурсивного, но он расходует больше памяти. Алгоритм Левенштейна работает быстрее модифицированной версии Дамерау-Левенштейна, т.к. в теле цикла выполняется меньше операций. Улучшение рекурсивного алгоритма, добавлением кэша, значительно увеличивает скорость работы рекурсивной реализации за счёт того, что не производятся повторные вычисления.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- [1] Comparing and Merging Files [Электронный ресурс]. Режим доступа: <https://www.gnu.org/software/diffutils/manual/diffutils.html> (дата обращения: 21.09.2022).
- [2] И. Левенштейн В. Двоичные коды с исправлением выпадений, вставок и замещений символов. – М.: «Наука», Доклады АН СССР, 1965. Т. 163. с. 845.
- [3] Язык программирования Go [Электронный ресурс]. Режим доступа: <https://go.dev> (дата обращения: 21.09.2022).
- [4] Документация по ЯП Go: бенчмарки [Электронный ресурс]. Режим доступа: <https://go.dev/doc/tutorial/add-a-test> (дата обращения: 21.09.2022).
- [5] Windows 11 [Электронный ресурс]. Режим доступа: <https://www.microsoft.com/en-us/windows?wa=wsignin1.0> (дата обращения: 21.09.2022).
- [6] Процессор AMD Ryzen™ 5 3550H [Электронный ресурс]. Режим доступа: <https://www.amd.com/ru/products/apu/amd-ryzen-5-3550h> (дата обращения: 21.09.2022).