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

Тема Расстояние Левенштейна и Дамерау-Левенштейна

Студент Кононенко С.С.

Группа ИУ7-53Б

Оценка (баллы) _____

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

Москва — 2020 г.

Оглавление

Введение

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

Впервые задачу поставил в 1965 году советский математик Владимир Левенштейн при изучении последовательностей 0–1, впоследствии более общую задачу для произвольного алфавита связали с его именем.

Расстояние Левенштейна и его обобщения активно применяются:

1) для исправления ошибок в слове (в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи);

2) для сравнения текстовых файлов утилитой `diff` и ей подобными (здесь роль «символов» играют строки, а роль «строк» — файлы);

3) в биоинформатике для сравнения генов, хромосом и белков.

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: к операциям вставки, удаления и замены символов, определённых в расстоянии Левенштейна добавлена операция транспозиции (перестановки) символов.

Задачи работы

- Изучение алгоритмов Левенштейна и Дамерау–Левенштейна.

- Применение методов динамического программирования для реализации алгоритмов.
- Получение практических навыков реализации алгоритмов Левенштейна и Дамерау — Левенштейна.
- Сравнительный анализ алгоритмов на основе экспериментальных данных.
- Подготовка отчета по лабораторной работе.

1 Аналитическая часть

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

Цены операций могут зависеть от вида операции (вставка, удаление, замена) и/или от участвующих в ней символов, отражая разную вероятность разных ошибок при вводе текста, и т. п. В общем случае:

- $w(a, b)$ — цена замены символа a на символ b.
- $w(\lambda, b)$ — цена вставки символа b.
- $w(a, \lambda)$ — цена удаления символа a.

Для решения задачи о редакционном расстоянии необходимо найти последовательность замен, минимизирующую суммарную цену. Расстояние Левенштейна является частным случаем этой задачи при

- $w(a, a) = 0$.
- $w(a, b) = 1,\ a \neq b$.
- $w(\lambda, b) = 1$.
- $w(a, \lambda) = 1$.

1.1 Рекурсивный алгоритм нахождения расстояния Левенштейна

Расстояние Левенштейна между двумя строками a и b может быть вычислено по формуле 1.1, где $|a|$ означает длину строки a; $a[i]$ — i-ый символ строки a , функция $D(i,j)$ определена как:

$$D(i,j) = \begin{cases} 0 & \text{i} = 0,\ \text{j} = 0 \\ i & \text{j} = 0,\ \text{i} > 0 \\ j & \text{i} = 0,\ \text{j} > 0 \\ \min\{ & \\ \quad D(i,j-1)+1 & \\ \quad D(i-1,j)+1 & \text{i} > 0,\ \text{j} > 0 \\ \quad D(i-1,j-1)+m(a[i],b[j]) & (1.2) \\ \} & \end{cases}, \qquad (1.1)$$

а функция 1.2 определена как:

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

Рекурсивный алгоритм реализует формулу 1.1. Функция D составлена из следующих соображений:

1) для перевода из пустой строки в пустую требуется ноль операций;

2) для перевода из пустой строки в строку a требуется $|a|$ операций;

3) для перевода из строки a в пустую требуется $|a|$ операций;

4) для перевода из строки a в строку b требуется выполнить последовательно некоторое количество операций (удаление, вставка, замена) в некоторой последовательности. Последовательность проведения любых двух операций можно поменять, порядок проведения

операций не имеет никакого значения. Полагая, что a', b' — строки a и b без последнего символа соответственно, цена преобразования из строки a в строку b может быть выражена как:

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 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.

1.2 Матричный алгоритм нахождения расстояния Левенштейна

Прямая реализация формулы 1.1 может быть малоэффективна по времени исполнения при больших i, j, т. к. множество промежуточных значений $D(i, j)$ вычисляются заново множество раз подряд. Для оптимизации нахождения расстояния Левенштейна можно использовать матрицу в целях хранения соответствующих промежуточных значений. В таком случае алгоритм представляет собой построчное заполнение матрицы $A_{|a|,|b|}$ значениями $D(i, j)$.

1.3 Рекурсивный алгоритм нахождения расстояния Левенштейна с заполнением матрицы

Рекурсивный алгоритм заполнения можно оптимизировать по времени выполнения с использованием матричного алгоритма. Суть данного метода 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 Расстояния Дамерау — Левенштейна

Расстояние Дамерау — Левенштейна может быть найдено по формуле 1.3, которая задана как

$$d_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{если } \min(i,j) = 0, \\ \min\{ & \\ \quad d_{a,b}(i,j-1)+1 & \\ \quad d_{a,b}(i-1,j)+1 & \text{иначе} \\ \quad d_{a,b}(i-1,j-1)+m(a[i],b[j]) & \\ \quad d_{a,b}(i-2,j-2)+1 & \text{если } i,j>1; \\ & a[i]=b[j-1]; \\ & b[j]=a[i-1] \\ \} & \end{cases}, \tag{1.3}$$

Формула выводится по тем же соображениям, что и формула (1.1). Как и в случае с рекурсивным методом, прямое применение этой формулы неэффективно по времени исполнения, то аналогично методу из 1.3

производится добавление матрицы для хранения промежуточных значений рекурсивной формулы.

Вывод

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

Пусть S_1 и S_2 — две строки (длиной M и N соответственно) над некоторым алфавитом, тогда расстояние Левенштейна $d(S_1, S_2)$ можно подсчитать по рекуррентной формуле $d(S_1, S_2) = D(M, N)$, где

$$D(i,j) = \begin{cases} 0, & i = 0,\ j = 0 \\ i, & i > 0,\ j = 0 \\ j, & i = 0,\ j > 0 \\ \min\{ & \\ \quad D(i, j-1) + 1, & \\ \quad D(i-1, j) + 1, & i > 0,\ j > 0 \\ \quad D(i-1, j-1) + m(S_1[i], S_2[j]) & \\ \} & \end{cases} \tag{1.4}$$

2 Конструкторская часть

2.1 Схема алгоритма Левенштейна

На рисунке 2.1 приведена схема рекурсивного алгоритма Левенштейна.

На рисунке 2.2 приведена схема рекурсивного алгоритма Левенштейна с заполнением матрицы.

На рисунке 2.3 приведена схема матричного алгоритма Левенштейна.

2.2 Схема алгоритма Дамерау — Левенштейна

На рисунке 2.4 приведена схема матричного алгоритма Дамерау – Левенштейна.

Вывод

На основе теоретических данных, полученных из аналитического раздела были построены схемы требуемых алгоритмов.

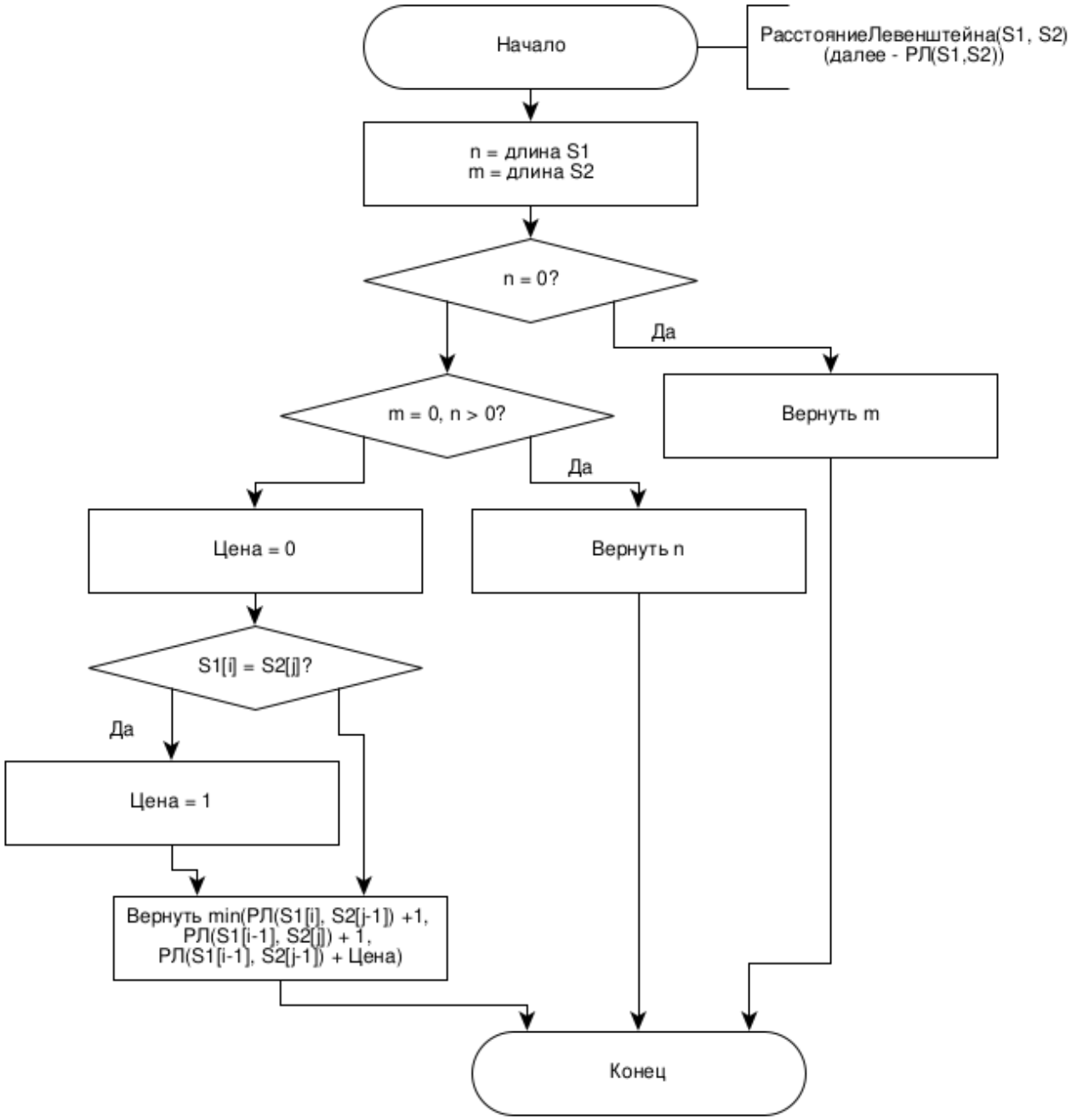

Рисунок 2.1: Схема рекурсивного алгоритма Левенштейна

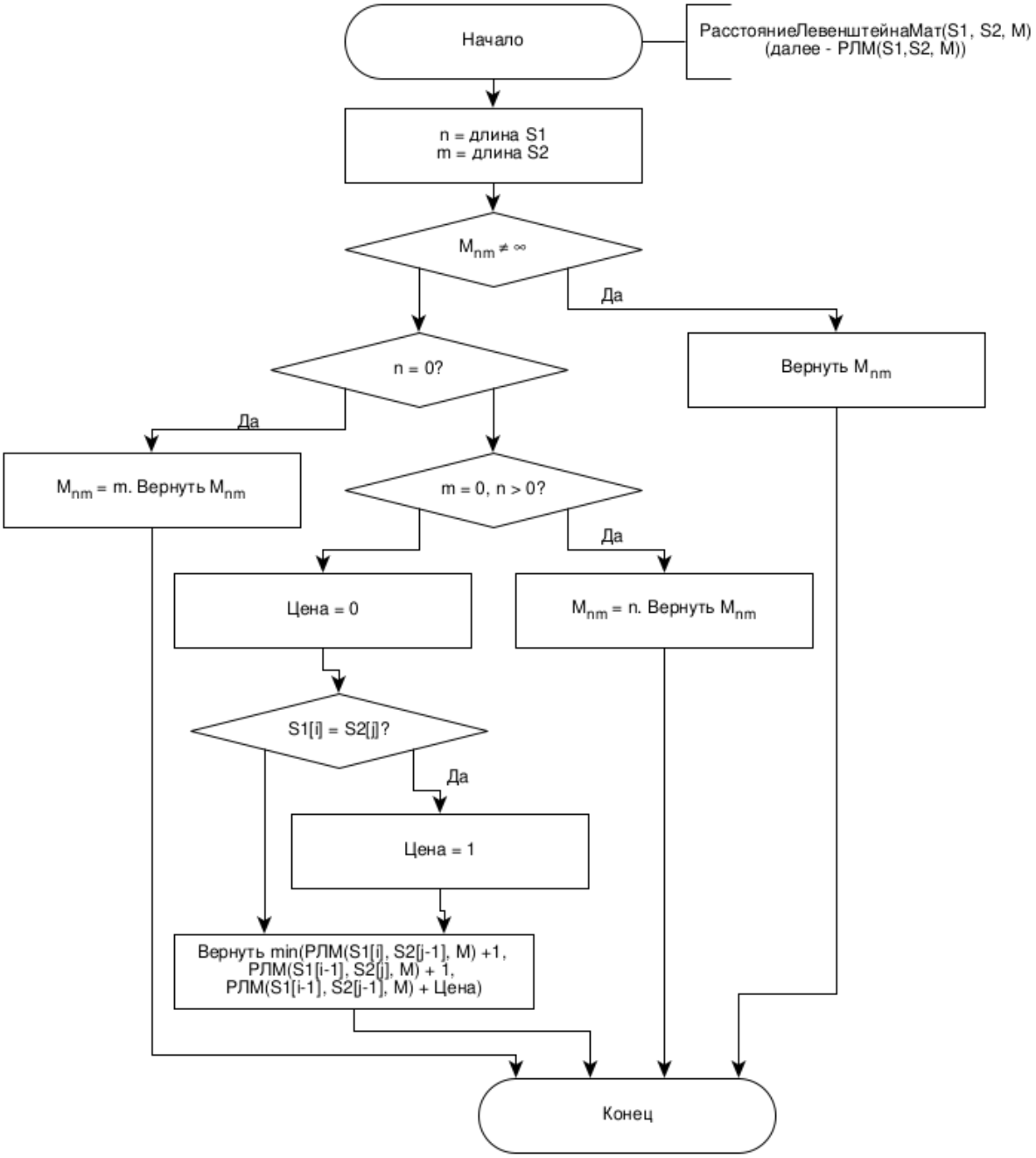

Рисунок 2.2: Схема рекурсивного алгоритма Левенштейна с заполнением матрицы

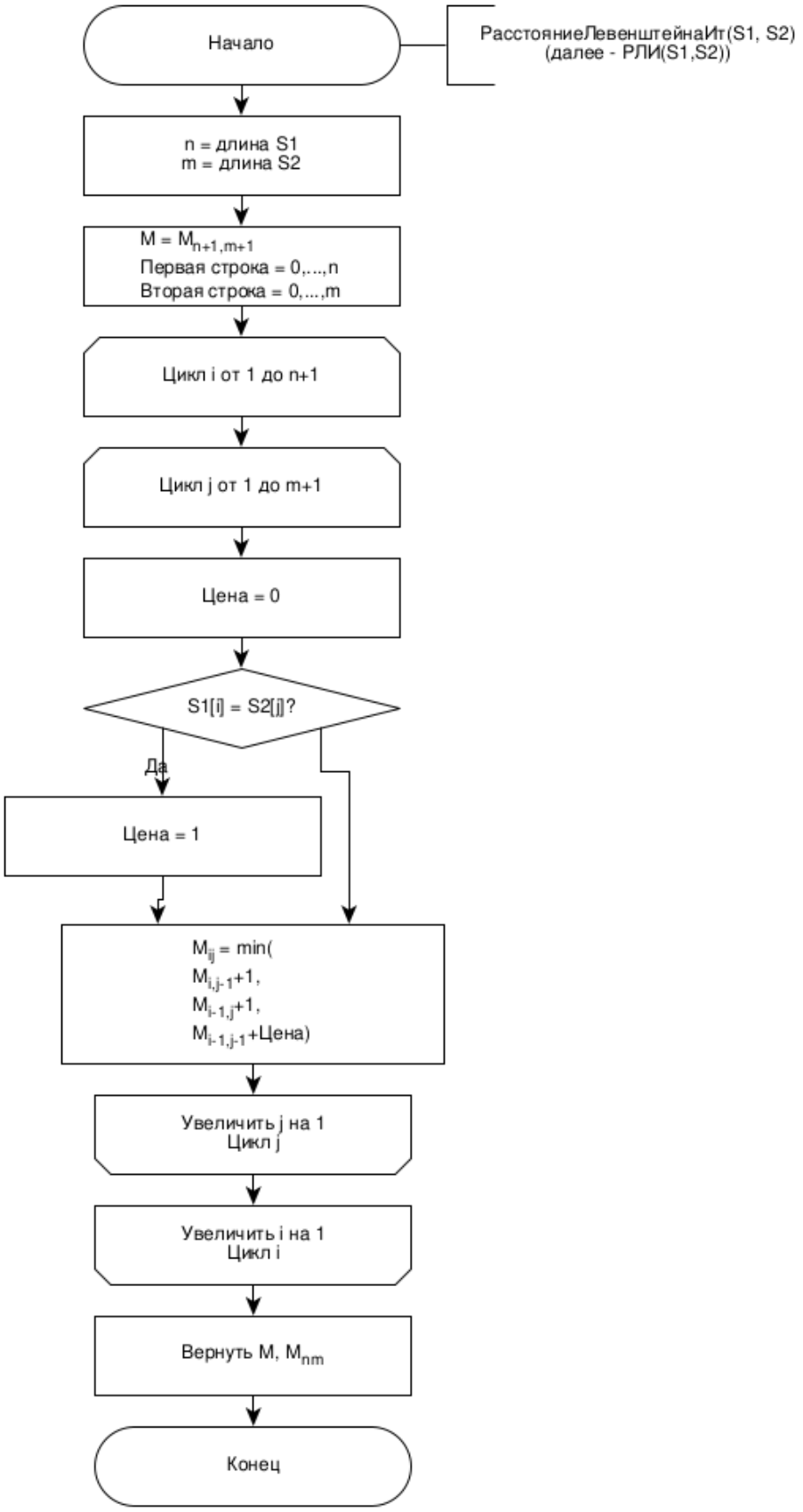

Рисунок 2.3: Схема матричного алгоритма Левенштейна

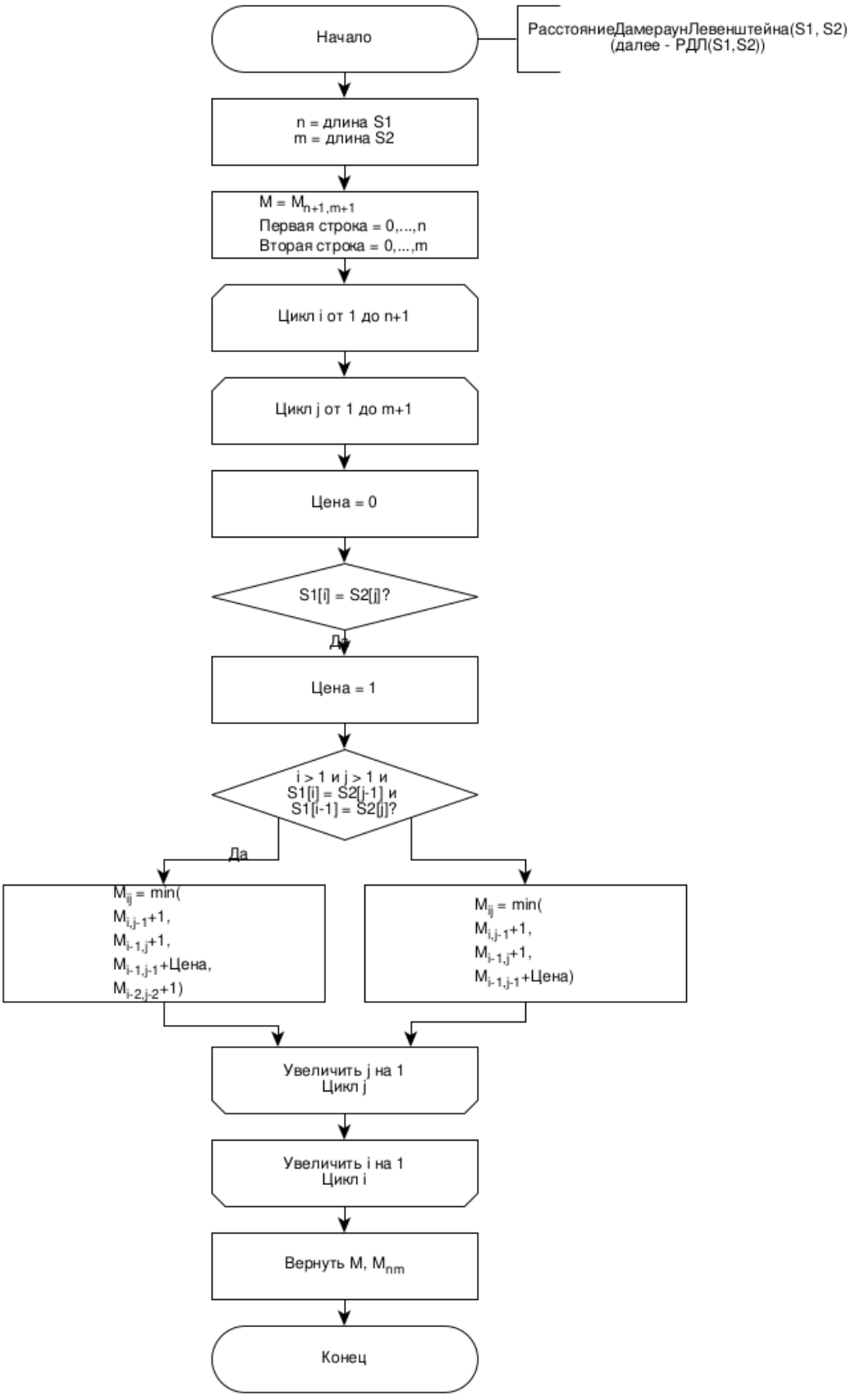

Рисунок 2.4: Схема алгоритма Дамерау – Левенштейна

3 Технологическая часть

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

3.1 Требования к ПО

К программе предъявляется ряд требований:

- на вход подаётся две строки в любой раскладке;
- на выходе — искомое расстояние для всех четырех методов и матрицы расстояний для всех методов, за исключением рекурсивного.

3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран многопоточный язык Golang [2]. Данный выбор обусловлен моим желанием расширить свои знания в области применения данного язкыа. Помимо этого, встроенные средства языка предоставляют высокоточные средства тестирования разработанного ПО.

3.3 Листинг кода

В листингах 3.1–3.3 приведены реализации алгоритмов Левенштейна и Дамерау — Левенштейна, а также вспомогательные функции.

```
1  package levenshtein
2
3  import "math"
4
5  // Recursive used to find Levenshtein distance with recursive method.
6  func Recursive(fWord, sWord string) int {
7     fWordRune, sWordRune := []rune(fWord), []rune(sWord)
8
9     n, m := len(fWordRune), len(sWordRune)
10
```

```
    return getDistance(fWordRune, sWordRune, n, m)
}

// RecursiveMatrix used to find Levenshtein distance with recursive method
    and matrix filling.
func RecursiveMatrix(fWord, sWord string) (int, MInt) {
    var (
        n, m, shDist int
    )

    fWordRune, sWordRune := []rune(fWord), []rune(sWord)

    n, m = len(fWordRune), len(sWordRune)

    distMat := makeMatrixRec(n, m)

    getDistanceRec(fWordRune, sWordRune, n, m, distMat)

    shDist = distMat[n][m]

    return shDist, distMat
}

// IterativeMatrix used to find Levenshtein distance with matrix filling.
func IterativeMatrix(fWord, sWord string) (int, MInt) {
    var (
        n, m, dist, shDist int
    )

    fWordRune, sWordRune := []rune(fWord), []rune(sWord)

    n, m = len(fWordRune), len(sWordRune)

    distMat := makeMatrix(n, m)

    for i := 1; i < n+1; i++ {
        for j := 1; j < m+1; j++ {
            insDist := distMat[i][j-1] + 1
            delDist := distMat[i-1][j] + 1
            eq := 1
            if fWordRune[i-1] == sWordRune[j-1] {
                eq = 0
            }
            eqDist := distMat[i-1][j-1] + eq

            dist = minFromThree(insDist, delDist, eqDist)
            distMat[i][j] = dist
        }
```

```
    }

    shDist = distMat[n][m]

    return shDist, distMat
}

// DamerauLevenshtein used to find Damerau-Levenshtein distance.
func DamerauLevenshtein(fWord, sWord string) (int, MInt) {
    var (
        n, m, dist, shDist, transDist int
    )

    fWordRune, sWordRune := []rune(fWord), []rune(sWord)

    n, m = len(fWordRune), len(sWordRune)

    distMat := makeMatrix(n, m)

    for i := 1; i < n+1; i++ {
        for j := 1; j < m+1; j++ {
            insDist := distMat[i][j-1] + 1
            delDist := distMat[i-1][j] + 1
            eq := 1
            if fWordRune[i-1] == sWordRune[j-1] {
                eq = 0
            }
            eqDist := distMat[i-1][j-1] + eq
            transDist = -1
            if i > 1 && j > 1 {
                transDist = distMat[i-2][j-2] + 1
            }

            if transDist != -1 && fWordRune[i-1] == sWordRune[j-2] &&
                fWordRune[i-2] == sWordRune[j-1] {
                dist = minFromFour(insDist, delDist, eqDist, transDist)
            } else {
                dist = minFromThree(insDist, delDist, eqDist)
            }
            distMat[i][j] = dist
        }
    }

    shDist = distMat[n][m]

    return shDist, distMat
}

```

```
105 func getDistanceRec(fWord, sWord []rune, i, j int, mat MInt) int {
106     if mat[i][j] != math.MaxInt16 {
107         return mat[i][j]
108     }
109 
110     if i == 0 {
111         mat[i][j] = j
112         return mat[i][j]
113     }
114     if j == 0 && i > 0 {
115         mat[i][j] = i
116         return mat[i][j]
117     }
118     eq := 1
119     if fWord[i-1] == sWord[j-1] {
120         eq = 0
121     }
122 
123     mat[i][j] = minFromThree(
124         getDistanceRec(fWord, sWord, i, j-1, mat)+1,
125         getDistanceRec(fWord, sWord, i-1, j, mat)+1,
126         getDistanceRec(fWord, sWord, i-1, j-1, mat)+eq)
127     return mat[i][j]
128 }
129 
130 func makeMatrixRec(n, m int) MInt {
131     mat := make(MInt, n+1)
132     for i := range mat {
133         mat[i] = make([]int, m+1)
134     }
135 
136     for i := 0; i < n+1; i++ {
137         for j := 0; j < m+1; j++ {
138             mat[i][j] = math.MaxInt16
139         }
140     }
141 
142     return mat
143 }
144 
145 func getDistance(fWord, sWord []rune, i, j int) int {
146     if i == 0 {
147         return j
148     }
149     if j == 0 && i > 0 {
150         return i
151     }
152     eq := 1
```

```
153     if fWord[i-1] == sWord[j-1] {
154         eq = 0
155     }
156 
157     return minFromThree(
158         getDistance(fWord, sWord, i, j-1)+1,
159         getDistance(fWord, sWord, i-1, j)+1,
160         getDistance(fWord, sWord, i-1, j-1)+eq)
161 }
162 
163 func makeMatrix(n, m int) MInt {
164     mat := make(MInt, n+1)
165     for i := range mat {
166         mat[i] = make([]int, m+1)
167     }
168 
169     for i := 0; i < m+1; i++ {
170         mat[0][i] = i
171     }
172 
173     for i := 0; i < n+1; i++ {
174         mat[i][0] = i
175     }
176 
177     return mat
178 }
179 
180 func minFromFour(a, b, c, d int) int {
181     min := a
182 
183     if b < min {
184         min = b
185     }
186 
187     if c < min {
188         min = c
189     }
190 
191     if d < min {
192         min = d
193     }
194 
195     return min
196 }
197 
198 func minFromThree(a, b, c int) int {
199     min := a
200 
```

```
    if b < min {
        min = b
    }

    if c < min {
        min = c
    }

    return min
}
```

Листинг 3.1: Функции реализации алгоритмов Левенштейна и Дамерау – Левенштейна

```
package levenshtein

import "fmt"

// MInt used to represent int64 matrix
type MInt [][]int

// PrintMatrix used to pretty print matrix.
func (mat MInt) PrintMatrix() {
    for i := 0; i < len(mat); i++ {
        for j := 0; j < len(mat[0]); j++ {
            fmt.Printf("%3d ", mat[i][j])
        }
        fmt.Printf("\n")
    }
}
```

Листинг 3.2: Пользовательские типы данных

```
package levenshtein

import "fmt"

// ReadWord used to read a word through EOL symbol.
func ReadWord() string {
    var word string
    fmt.Scanln(&word)

    return word
}
```

Листинг 3.3: Дополнительные утилиты

Строка 1	Строка 2	Ожидаемый результат	
		Левенштейн	Дамерау — Левенштейн
cook	cooker	2	2
mother	money	3	3
woman	water	4	4
program	friend	6	6
house	girl	5	5
probelm	problem	2	1
head	ehda	3	2
bring	brought	4	4
happy	happy	0	0
minute	moment	5	5
person	eye	5	5
week	weeks	1	1
member	morning	6	6
death	health	2	2
education	question	4	4
room	moor	2	2
car	city	3	3
air	area	3	3

Таблица 3.1: Функциональные тесты

В таблице 3.1 приведены функциональные тесты для алгоритмов вычисления расстояния Левенштейна и Дамерау — Левенштейна. Все тесты пройдены успешно.

Вывод

Были разработаны и протестированы схемы четырех алгоритмов: вычисления расстояния Левенштейна рекурсивно, с заполнением матрицы и рекурсивно с заполнением матрицы, а также вычисления расстояния Дамерау — Левенштейна с заполнением матрицы.

4 Исследовательская часть

4.1 Пример работы

Демонстрация работы программы приведена на рисунке 4.1.

```
~/bmstu/labs/aa-5th-sem-labs/lab_01/src > master !2 ?1  ./app.exe
Расстояние Левенштейна

Введите первое слово: сито
Введите второе слово: столб

Рекурсивный метод без заполнения матрицы:
Расстояние: 3


Рекурсивный метод с заполнением матрицы:
  0   1   2   3   4   5
  1   0   1   2   3   4
  2   1   1   2   3   4
  3   2   1   2   3   4
  4   3   2   1   2   3
Расстояние: 3


Итеративный метод с заполнением матрицы:
  0   1   2   3   4   5
  1   0   1   2   3   4
  2   1   1   2   3   4
  3   2   1   2   3   4
  4   3   2   1   2   3
Расстояние: 3


Метод Дамерау - Левенштейна:
  0   1   2   3   4   5
  1   0   1   2   3   4
  2   1   1   2   3   4
  3   2   1   2   3   4
  4   3   2   1   2   3
Расстояние: 3
```

Рисунок 4.1: Демонстрация работы алгоритмов нахождения расстояния Левенштейна и Дамерау – Левенштейна

4.2 Технические характеристики

- Операционная система: Manjaro [3] Linux [4] 20.1 64-bit.
- Память: 16 GiB.
- Процессор: AMD Ryzen™ 7 3700U [5] CPU @ 2.30GHz

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

4.3 Время выполнения алгоритмов

Алгоритмы тестировались при помощи написания «бенчмарков» [6], предоставялемых встроенными в Go средствами. Для такого рода тестирования не нужно самостоятельно указывать количество повторов. В библиотеке для тестирования существует константа N, которая динамически варьируется в зависимости от того, был ли получен стабильный результат или нет.

В листинге 4.1 пример реализации бенчмарка.

```
1  package levenshtein
2
3  import (
4      "testing"
5  )
6
7  // Recursive method benchmarks.
8
9  func BenchmarkRecursiveLen5(b *testing.B) {
10     fWord := "about"
11     sWord := "above"
12
13     for i := 0; i < b.N; i++ {
14         Recursive(fWord, sWord)
15     }
16 }
```

Листинг 4.1: Реализация бенчмарка

Результаты замеров приведены в таблице 4.1. На рисунках 4.2 и 4.3 приведены графики зависимостей времени работы алгоритмов от длины строк.

Длина строк	Время, нс			
	Rec	**RecMat**	**ItMat**	**DL**
5	16486	NaN	NaN	NaN
10	79380187	6616	2104	2532
15	300320151997	NaN	NaN	NaN
20	NaN	23761	6649	7952
30	NaN	50696	13700	16482
50	NaN	134389	36555	42789
100	NaN	562996	139671	164096
200	NaN	2245510	550606	645347

Таблица 4.1: Замер времени для строк, размеромот 5 до 200

4.4 Использование памяти

Алгоритмы Левенштейна и Дамерау — Левенштейна не отличаются друг от друга с точки зрения использования памяти, следовательно, достаточно рассмотреть лишь разницу рекурсивной и матричной реализаций этих алгоритмов.

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк, соответственно, максимальный расход памяти (4.1)

$$(\mathcal{C}(S_1) + \mathcal{C}(S_2)) \cdot (2 \cdot \mathcal{C}(\text{string}) + 3 \cdot \mathcal{C}(\text{int})), \tag{4.1}$$

где $\mathcal{C}$ — оператор вычисления размера, S_1, S_2 — строки, int — целочисленный тип, string — строковый тип.

Использование памяти при итеративной реализации теоретически равно

$$(\mathcal{C}(S_1) + 1) \cdot (\mathcal{C}(S_2) + 1) \cdot \mathcal{C}(\text{int}) + 10 \cdot \mathcal{C}(\text{int}) + 2 \cdot \mathcal{C}(\text{string}). \tag{4.2}$$

Выделение памяти при работе алгоритмов указано на рисунке 4.4.

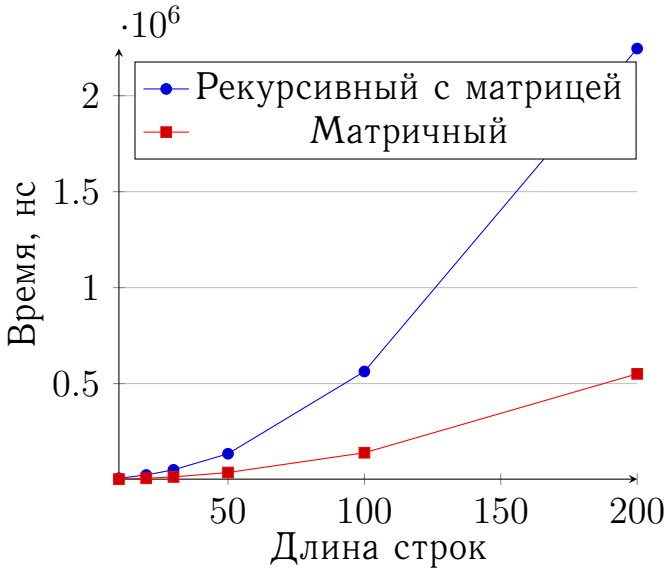

Рисунок 4.2: Зависимость времени работы алгоритма вычисления расстояния Левенштейна от длины строк (рекурсивная с заполнением матрицы и матричная реализации)

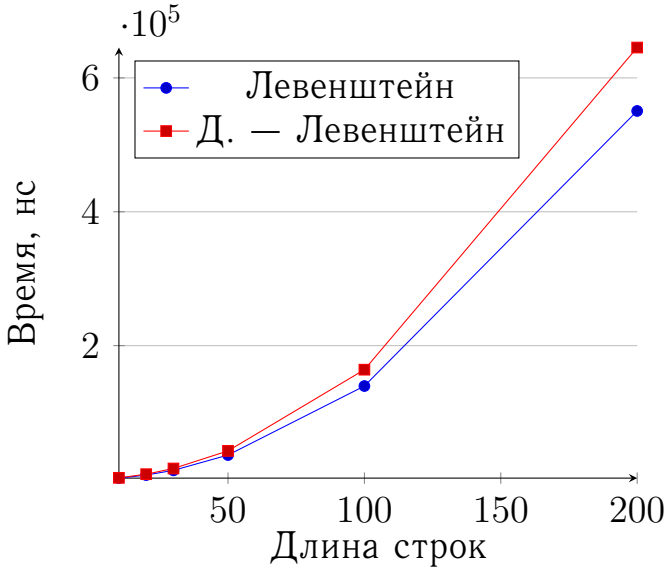

Рисунок 4.3: Зависимость времени работы матричных реализаций алгоритмов Левенштейна и Дамерау — Левенштейна

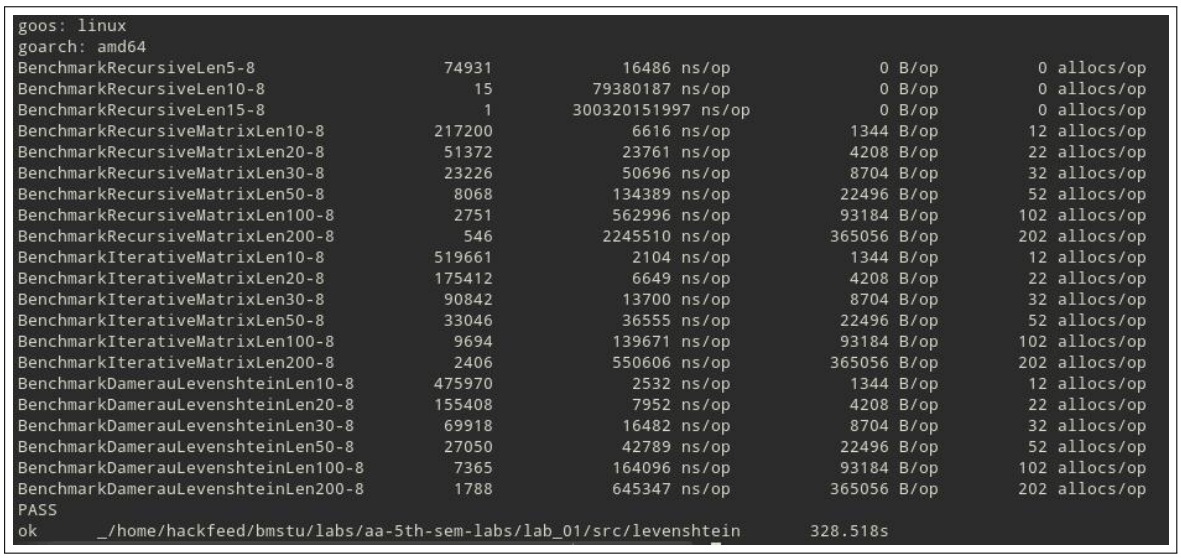

Рисунок 4.4: Замеры производительности алгоритмов, выполненные при помощи команды `go test -becnh . -benchmem`

Вывод

Рекурсивный алгоритм Левенштейна работает на порядок дольше итеративных реализаций, время его работы увеличивается в геометрической прогрессии. На словах длиной 10 символов, матричная реализация алгоритма Левенштейна превосходит по времени работы рекурсивную в 37000 раз. Рекурсивный алгоритм с заполнением матрицы превосходит простой рекурсивный на аналогичных данных в 12000 раз. Алгоритм Дамерау — Левенштейна по времени выполнения сопоставим с алгоритмом Левенштейна. В нём добавлены дополнительные проверки, и по сути он является алгоритмом другого смыслового уровня.

Но по расходу памяти итеративные 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 работы были выполнены все поставленные задачи и изучены методы динамического программирования на основе алгоритмов вычисления расстояния Левенштейна.

Экспериментально были установлены различия в производительности различных алгоритмов вычисления расстояния Левенштейна. Рекурсивный алгоритм Левенштейна работает на несколько порядков медленнее (37000 раз) матричной реализации. Рекурсивный алгоритм с параллельным заполнением матрицы работает быстрее простого рекурсивного, но все еще медленнее матричного (12000 раз). Если длина сравниваемых строк превышает 10, рекурсивный алгоритм становится неприемлимым для использования по времени выполнения программы. Матричная реализация алгоритма Дамерау — Левенштейна сопоставимо с алгоритмом Левенштейна. В ней добавлены дополнительные проверки, но, эти алгоритмы находятся в разном поле использования.

Теоретически было рассчитано использования памяти в каждом из алгоритмов вычисления расстояния Левенштейна. Обычные матричные алгоритмы потребляют намного больше памяти, чем рекурсивные, за счет дополнительного выделения памяти под матрицы и большее количество локальных переменных.

Литература

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

[2] The Go Programming Language. Режим доступа: `https://golang.org/`. Дата обращения: 11.09.2020.

[3] Manjaro – enjoy the simplicity. Режим доступа: `https://manjaro.org/`. Дата обращения: 14.09.2020.

[4] Linux – Википедия. Режим доступа: `https://ru.wikipedia.org/wiki/Linux`. Дата обращения: 14.09.2020.

[5] Мобильный процессор AMD Ryzen™ 7 3700U с графикой Radeon™ RX Vega 10. Режим доступа: `https://www.amd.com/ru/products/apu/amd-ryzen-7-3700u`. Дата обращения: 14.09.2020.

[6] testing – The Go Programming Language. Режим доступа: `https://golang.org/pkg/testing/`. Дата обращения: 12.09.2020.