



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, искусственный интеллект и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Отчет по лабораторной работе №1 по курсу "Анализ алгоритмов"

Тема Расстояние Левенштейна и Дamerau-Левенштейна

Студент Алькина А.Р.

Группа ИУ7-54Б

Оценка (баллы) _____

Преподаватели Волкова Л.Л., Строганов Д.В.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	3
1 Аналитическая часть	4
1.1 Рекурсивный алгоритм нахождения расстояния Левенштейна	5
1.2 Матричный алгоритм нахождения расстояния Левенштейна	7
1.3 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна .	8
1.4 Матричный алгоритм нахождения расстояния Дамерау-Левенштейна . .	9
1.5 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна с заполнением матрицы	10
2 Конструкторская часть	11
2.1 Схема алгоритма Левенштейна	11
2.2 Схема алгоритма Дамерау-Левенштейна	11
3 Технологическая часть	17
3.1 Требования к ПО	17
3.2 Средства реализации	17
3.3 Листинг кода	17
3.4 Функциональное тестирование	24
4 Исследовательская часть	25
4.1 Пример работы	25
4.2 Технические характеристики	26
4.3 Время выполнения алгоритмов	26
4.4 Использование памяти	27
ЗАКЛЮЧЕНИЕ	30
СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ	31

ВВЕДЕНИЕ

Цель лабораторной работы – изучение, анализ и реализация алгоритмов нахождения расстояний между строками Левенштейна и Дameraу-Левенштейна.

Расстояние Левенштейна – минимальное количество редакторских операций вставки, удаления и замены символа, которое необходимо для преобразования одной строки в другую.

Впервые задачу поставил в 1965 году советский математик Владимир Левенштейн [1] при изучении последовательностей $0-1$, впоследствии более общую задачу для произвольного алфавита связали с его именем. Большой вклад в изучение вопроса внёс Дэн Гасфилд.

Расстояние Левенштейна применяется в:

1. Компьютерной лингвистике для исправления ошибок.
2. Для сравнения текстовых файлов утилитой `diff` и другими.
3. В биоинформатике для сравнения генов, хромосом и белков.

Расстояние Дameraу-Левенштейна (названо в честь учёных Фредерика Дameraу и Владимира Левенштейна) – минимальное количество редакторских операций вставки, удаления, замены символа или транспозиции символов, которое необходимо для преобразования одной строки в другую.

Задачи лабораторной работы:

1. Изучение алгоритмов Левенштейна и Дameraу-Левенштейна.
2. Применение методов динамического программирования для реализации алгоритмов.
3. Получение практических навыков реализации алгоритмов Левенштейна и Дameraу-Левенштейна.
4. Сравнительный анализ алгоритмов на основе экспериментальных данных.
5. Подготовка отчета по лабораторной работе.

1 Аналитическая часть

Чтобы вычислить расстояние Левенштейна вводятся редакционные предписания – последовательность действий, необходимых для получения из первой строки второй кратчайшим образом. Обычно действия обозначаются так: D (англ. delete) – удалить, I (англ. insert) – вставить, R (replace) – заменить, M (match) – совпадение.

Цены операций могут зависеть от вида операции (вставка, удаление, замена) и/или от участвующих в ней символов, отражая разную вероятность мутаций в биологии, разную вероятность разных ошибок при вводе текста и т. д. 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 редакционных предписаний, 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

Существует проблема взаимного выравнивания строк.

Таблица 1.1 — Пример неоптимального выбора расположения строк

s1	у	в	л	е	ч	е	н	и	е		
s2	р	а	з	в	л	е	ч	е	н	и	е
Операции	R	R	R	R	R	M	R	R	R	I	I

Сумма цен операций равна 10.

Теперь расположим строки относительно друг друга иным образом.

Таблица 1.2 – Пример оптимального выбора расположения строк

s1			у	в	л	е	ч	е	н	и	е
s2	р	а	з	в	л	е	ч	е	н	и	е
Операции	I	I	R	M	M	M	M	M	M	M	M

Сумма цен операций равна 3, что и является расстоянием Левенштейна для данных строк.

Проблема решается введением рекуррентной формулы.

Обозначим:

- $L1$ — длина $S1$ (первой строки);
- $L2$ — длина $S2$ (второй строки);
- $S1[1...i]$ — подстрока $S1$ длиной i , начиная с первого символа;
- $S2[1...j]$ — подстрока $S2$ длиной j , начиная с первого символа;
- $S1[i]$ — i -й символ строки $S1$;
- $S2[j]$ — j -й символ строки $S2$.

Тогда расстояние Левенштейна между двумя строками $S1$ и $S2$ может быть вычислено по формуле 1.3, где функция $D(S1[1...i], S2[1...j])$ определена как:

$$D(S1[1...i], S2[1...j]) = \begin{cases} 0 & i = 0, j = 0, \\ j & i = 0, j > 0, \\ i & i > 0, j = 0, \\ \min\{ & \\ \quad D(S1[1...i], S2[1...j-1]) + 1 & \\ \quad D(S1[1...i-1], S2[1...j]) + 1 & i > 0, j > 0 \\ \quad D(S1[1...i-1], S2[1...j-1]) + & \\ \quad + f(S1[i], S2[j]) & (1.4) \\ \} & \end{cases} \quad (1.1)$$

а функция 1.4 определена как:

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

Рекурсивный алгоритм реализует формулу 1.3. Функция D составлена исходя 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 (вставка), где $|a|$ — длина строки a .
3. для перевода из строки a в пустую требуется $|a|$ операций (удаление).

Для перевода из строки $S1$ в строку $S2$ требуется выполнить последовательно некоторое количество операций (удаление, вставка, замена) в некоторой последовательности. Последовательность проведения любых двух операций можно менять, порядок проведения операций не важен. Если $S1', S2'$ — строки $S1$ и $S2$ без последнего символа соответственно, то цена преобразования из строки $S1$ в строку $S2$ может быть выражена как:

1. Сумма цены преобразования строки $S1$ в $S2$ и цены проведения операции удаления (1), которая необходима для преобразования $S1'$ в $S1$.
2. Сумма цены преобразования строки $S1$ в $S2$ и цены проведения операции вставки (1), которая необходима для преобразования $S2'$ в $S2$.
3. Сумма цены преобразования из $S1'$ в $S2'$ и цены проведения операции замены (1), предполагая, что $S1$ и $S2$ оканчиваются разные символы.
4. Цена преобразования из $S1'$ в $S2'$ при условии, что $S1$ и $S2$ оканчиваются на один и тот же символ.

Минимальной ценой преобразования будет минимальное значение приведенных вариантов.

1.2 Матричный алгоритм нахождения расстояния Левенштейна

Альтернативным подходом для более эффективного вычисления Левенштейна является использование матрицы для сохранения промежуточных значений. Этот подход позволяет избежать повторных вычислений множества значений $D(S1[1...i], S2[1...j])$ при больших значениях i и j . Вместо этого алгоритм заполняет матрицу $A_{|S1|+1, |S2|+1}$ построчно значениями $D(S1[1...i], S2[1...j])$.

Таким образом, вычисление значения $D(S1[1...i], S2[1...j])$ зависит от значений, сохраненных в матрице A . Вместо повторного вычисления, алгоритм может просто обращаться к матрице для получения необходимого значения. Это значительно сокращает количество вычислений и делает алгоритм более эффективным.

Искомое расстояние при этом будет находиться в элементе матрицы $A[i][j]$.

1.3 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна

Исходя из того, что 80% ошибок при наборе текста приходится на транспозицию (два соседних символа меняются местами), Фредерик Дамерау предложил ввести четвёртую редакторскую операцию – транспозицию (перестановку).

Обозначим:

- $L1$ – длина $S1$ (первой строки);
- $L2$ – длина $S2$ (второй строки);
- $S1[1...i]$ – подстрока $S1$ длиной i , начиная с первого символа;
- $S2[1...j]$ – подстрока $S2$ длиной j , начиная с первого символа;
- $S1[i]$ – i -й символ строки $S1$;
- $S2[j]$ – j -й символ строки $S2$.

Тогда расстояние Дамерау-Левенштейна между двумя строками $S1$ и $S2$ может быть вычислено по формуле 1.3, где функция $D(S1[1...i], S2[1...j])$

определена как:

$$D(S1[1...i], S2[1...j]) = \begin{cases} 0 & i = 0, j = 0, \\ j & i = 0, j > 0, \\ i & i > 0, j = 0, \\ \min\{ \\ \quad D(S1[1...i-1], S2[1...j-1]) + 1 \\ \quad D(S1[1...i], S2[1...j-1]) + 1 & \text{если } (i > 1, j > 1 \\ \quad D(S1[1...i-1], S2[1...j]) + 1 & S1[i] = S2[j-1], \\ \quad D(S1[1...i-1], S2[1...j-1]) + & S1[i-1] = S2[j]), \\ \quad + f(S1[i], S2[j]) & (1.4) \\ \}, \\ \min\{ \\ \quad D(S1[1...i], S2[1...j-1]) + 1 & \text{иначе,} \\ \quad D(S1[1...i-1], S2[1...j]) + 1 \\ \quad D(S1[1...i-1], S2[1...j-1]) + \\ \quad + f(S1[i], S2[j]) & (1.4) \\ \} \end{cases} \quad (1.3)$$

а функция 1.4 определена как:

$$f(a, b) = \begin{cases} 0 & \text{если } a = b, \\ 1 & \text{иначе.} \end{cases} \quad (1.4)$$

1.4 Матричный алгоритм нахождения расстояния Дамерау-Левенштейна

Матричный алгоритм нахождения расстояния Дамерау-Левенштейна реализуется практически идентично матричному алгоритму Левенштейна. Но в данном алгоритме также задействуется элемент матрицы $A[i-2][j-2]$, если возможна транспозиция символов на текущем шаге.

1.5 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна с заполнением матрицы

Рекурсивную реализацию алгоритма Дамерау-Левенштейна можно ускорить, комбинируя рекуррентный и матричный подход.

Для более эффективного вычисления расстояния Левенштейна параллельно можно заполнять матрицу, которая будет использоваться для кэширования уже вычисленных результатов рекурсивных вызовов. В процессе заполнения матрицы рекурсивно вызываются только те вычисления, которые еще не были обработаны. Результаты нахождения расстояния сохраняются в матрице, и если уже обработанные данные встречаются снова, расстояние для них не находится и алгоритм переходит к следующему шагу.

Вывод

Алгоритмы Левенштейна и Дамерау-Левенштейна для вычисления расстояния между строками могут быть реализованы как рекурсивно, так и итерационно. Оба алгоритма определяют расстояние между двумя строками путем вычисления минимального количества операций (вставки, удаления и замены символов, а также транспозиции), необходимых для преобразования одной строки в другую.

2 Конструкторская часть

2.1 Схема алгоритма Левенштейна

На рисунке 2.1 приведена схема матричного алгоритма Левенштейна.

2.2 Схема алгоритма Дамерау-Левенштейна

На рисунке 2.2 приведена схема рекурсивного алгоритма Дамерау-Левенштейна.

На рисунке 2.3 приведена схема рекурсивного алгоритма Дамерау-Левенштейна с заполнением матрицы. На дополнительном рисунке 2.4 приведена подпрограмма рекурсивного алгоритма Дамерау-Левенштейна с заполнением матрицы.

На рисунке 2.5 приведена схема матричного алгоритма Дамерау-Левенштейна.

Вывод

В данном разделе представлены схемы алгоритмов Левенштейна и Дамерау-Левенштейна, построенные в соответствии с теоретическими сведениями из аналитического раздела.

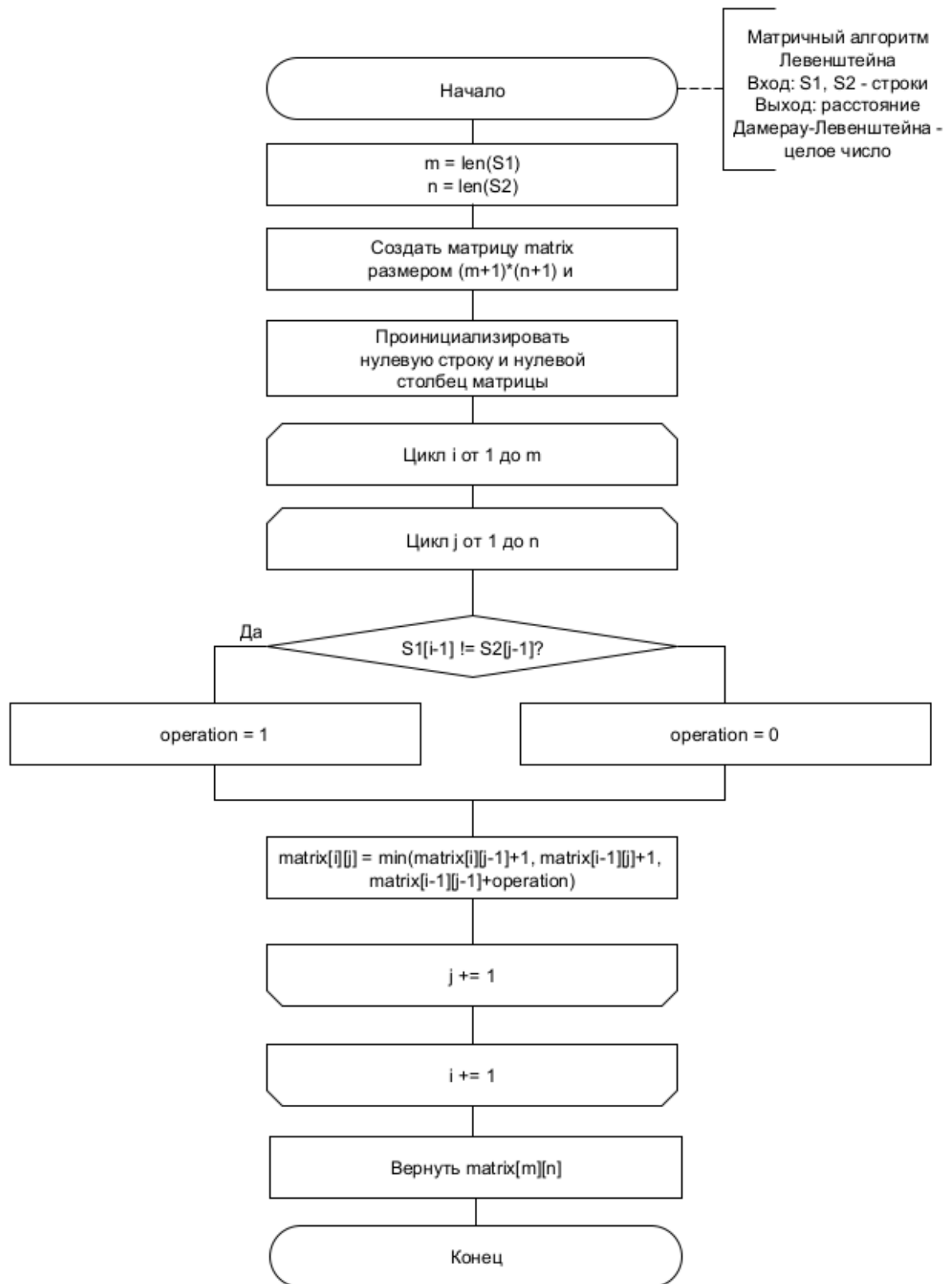


Рисунок 2.1 – Схема матричного алгоритма Левенштейна

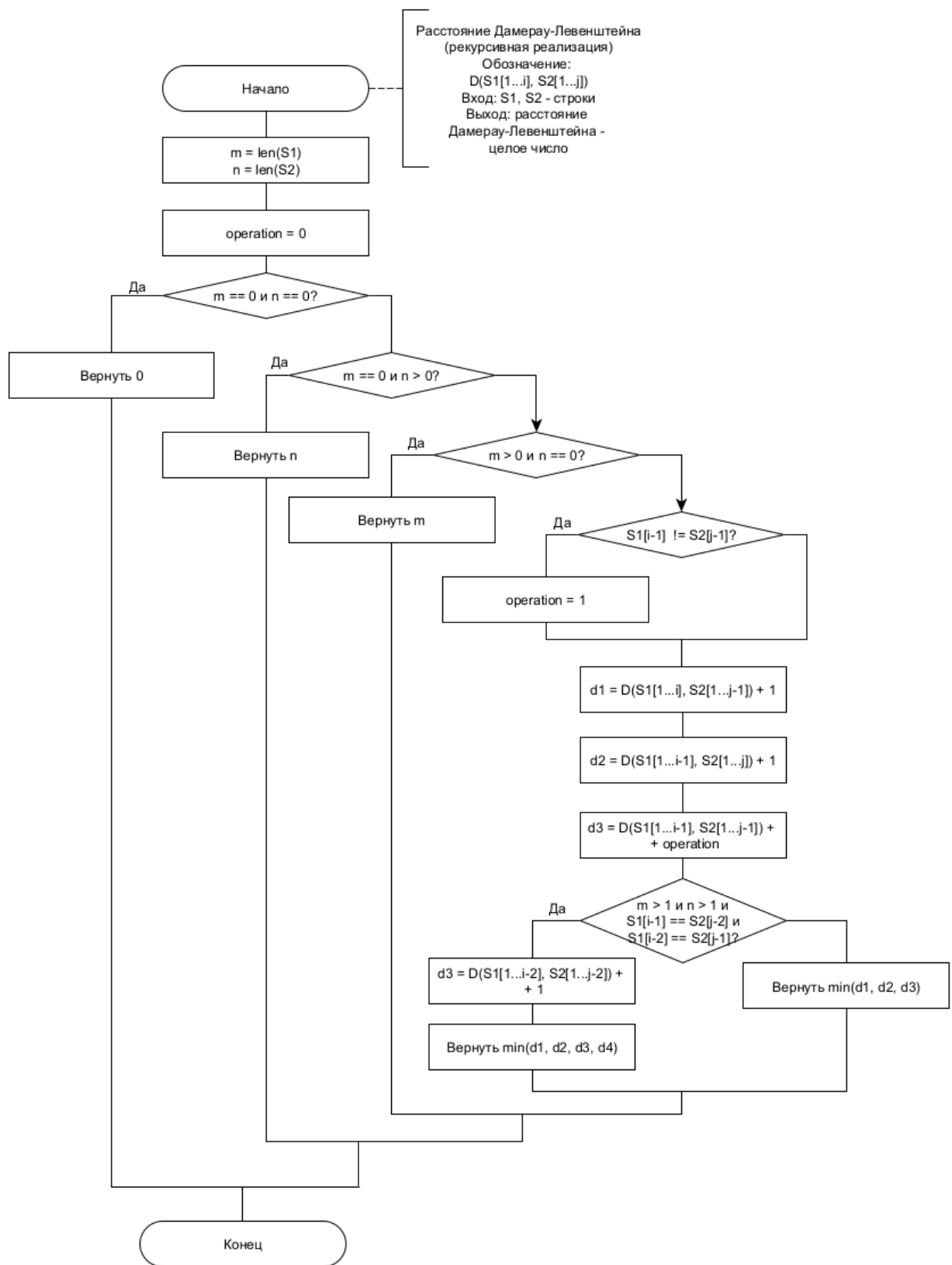


Рисунок 2.2 — Схема рекурсивного алгоритма Дамерау-Левенштейна

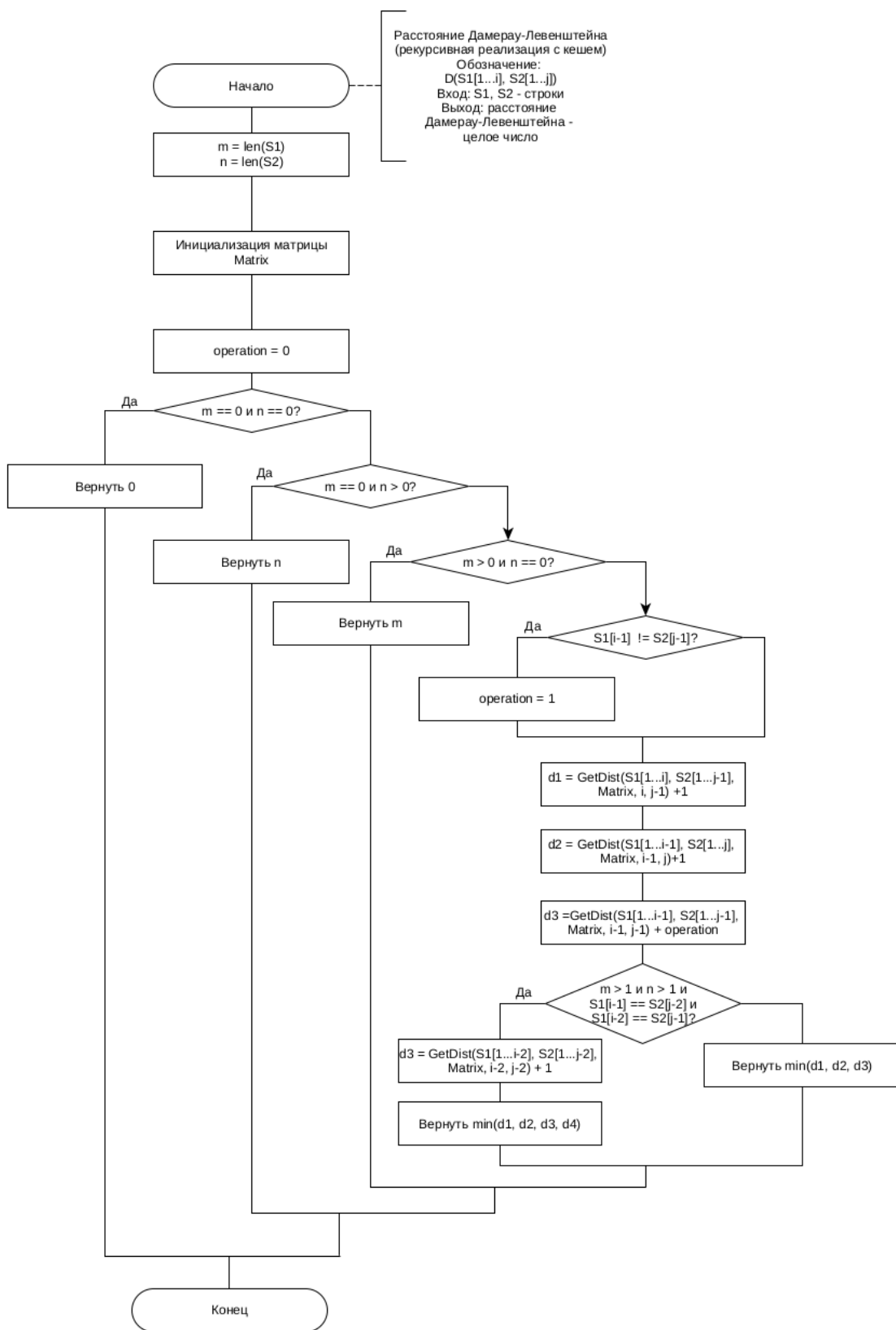


Рисунок 2.3 – Схема рекурсивного алгоритма Дамерау-Левенштейна с заполнением матрицы

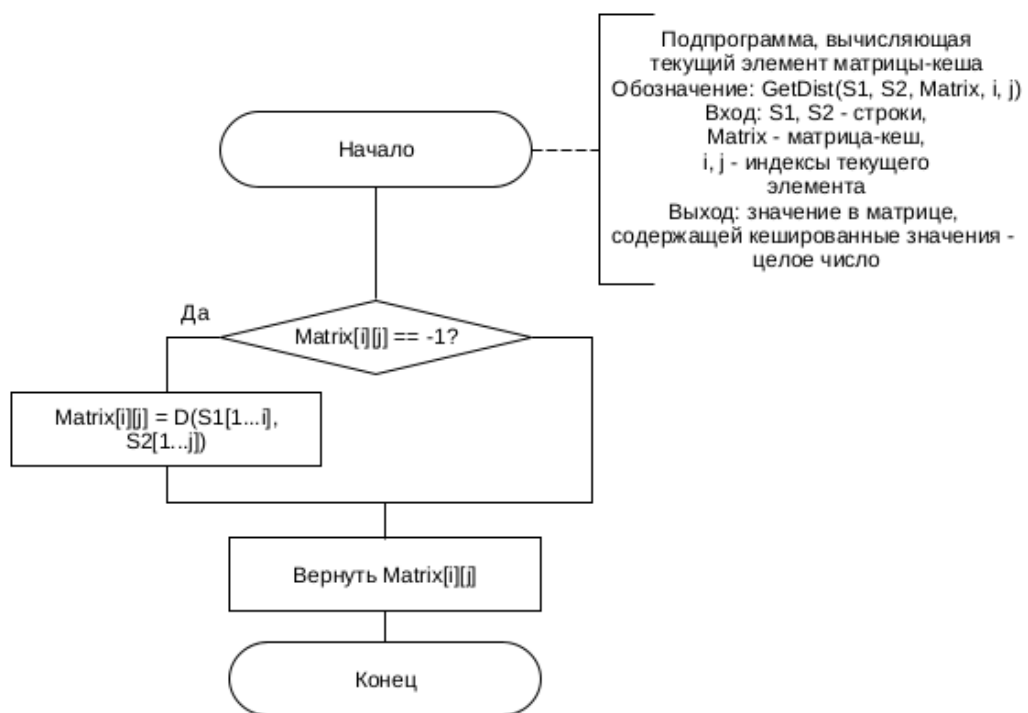


Рисунок 2.4 – Схема подпрограммы GetDist

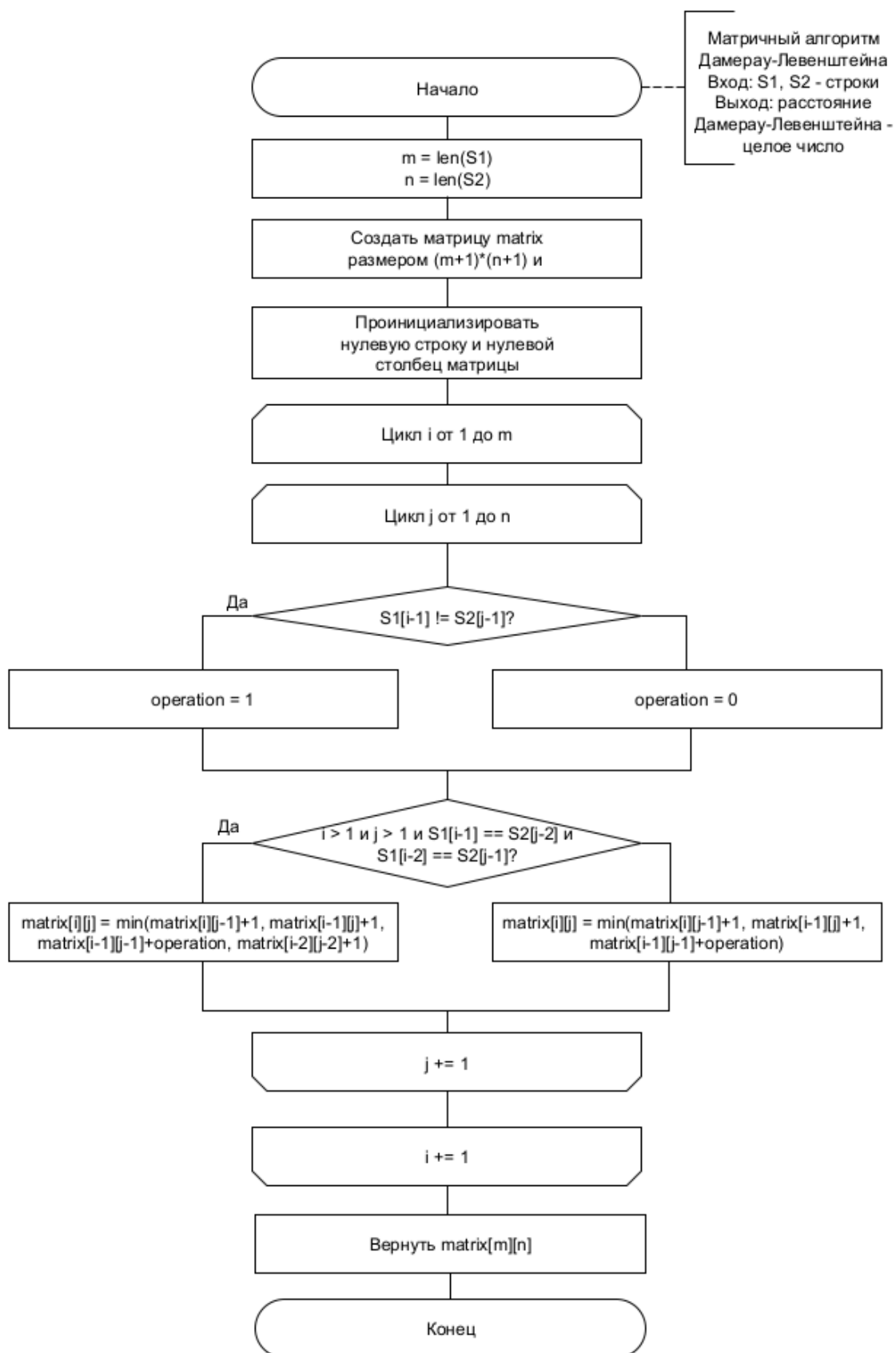


Рисунок 2.5 – Схема матричного алгоритма Дамерау-Левенштейна

3 Технологическая часть

В данном разделе представлены требования к программному обеспечению, а также рассматриваются средства реализации и приводятся листинги кода.

3.1 Требования к ПО

К программе предъявляется ряд требований:

- на вход подаются две строки;
- на выходе — искомое расстояние для всех четырех методов и матрицы расстояний для всех методов, которые подразумевают использование матрицы.

3.2 Средства реализации

В качестве языка программирования для реализации лабораторной работы был выбран многопоточный язык **GO** [2]. Выбор был сделан в пользу данного языка программирования, вследствие моего желания освоить данный язык программирования.

3.3 Листинг кода

В листингах 3.1–3.4 приведены реализации алгоритмов Левенштейна и Дamerau-Левенштейна.

Листинг 3.1 – Матричный (Левенштейн)

```
1 func MatrixLeven(str1 []rune, str2 []rune) (int, matrix.Matrix) {
2     operation := 0
3     matr := matrix.CreateMatrix(len(str1)+1, len(str2)+1, false)
4
5     for i := 1; i < matr.M; i++ {
6         for j := 1; j < matr.N; j++ {
7             if str1[i-1] != str2[j-1] {
8                 operation = 1
9             } else {
10                operation = 0
11            }
12            matr.Matr[i][j] = GetMin(matr.Matr[i][j-1]+1, matr.Matr[i-1][j]+1,
13                matr.Matr[i-1][j-1]+operation)
14        }
15    }
16    return matr.Matr[matr.M-1][matr.N-1], matr
17 }
```

Листинг 3.2 – Рекурсивный (Дамерау-Левенштейн)

```
1 func DamerauLeven(str1, str2 []rune) int {
2     m, n := len(str1), len(str2)
3     var operation, transposition int
4
5     if m == 0 && n == 0 {
6         return 0
7     } else if m == 0 && n > 0 {
8         return n
9     } else if m > 0 && n == 0 {
10        return m
11    } else {
12        if str1[m-1] != str2[n-1] {
13            operation = 1
14        }
15        moveLeft := DamerauLeven(str1, str2[:n-1]) + 1
16        moveRight := DamerauLeven(str1[:m-1], str2) + 1
17        moveDiag := DamerauLeven(str1[:m-1], str2[:n-1]) + operation
18        if m >= 2 && n >= 2 && str1[m-1] == str2[n-2] && str1[m-2] == str2[n-1] {
19            transposition = DamerauLeven(str1[:m-2], str2[:n-2]) + 1
20            return GetMin(moveLeft, moveRight, moveDiag, transposition)
21        }
22        return GetMin(moveDiag, moveRight, moveLeft)
23    }
24
25    return 0
26 }
```

Листинг 3.3 – Матричный (Дамерау-Левенштейн)

```
1 func MatrixDamerauLeven(str1 []rune, str2 []rune) (int, matrix.Matrix) {
2     operation := 0
3     matr := matrix.CreateMatrix(len(str1)+1, len(str2)+1, false)
4
5     for i := 1; i < matr.M; i++ {
6         for j := 1; j < matr.N; j++ {
7             if str1[i-1] != str2[j-1] {
8                 operation = 1
9             } else {
10                operation = 0
11            }
12            if i >= 2 && j >= 2 && str1[i-1] == str2[j-2] && str1[i-2] == str2[j-1] {
13                matr.Matr[i][j] = GetMin(matr.Matr[i][j-1]+1, matr.Matr[i-1][j]+1,
14                    matr.Matr[i-1][j-1]+operation, matr.Matr[i-2][j-2]+1)
15            } else {
16                matr.Matr[i][j] = GetMin(matr.Matr[i][j-1]+1, matr.Matr[i-1][j]+1,
17                    matr.Matr[i-1][j-1]+operation)
18            }
19        }
20    }
21    return matr.Matr[matr.M-1][matr.N-1], matr
22 }
```

Листинг 3.4 – Рекурсивный с кешем (Дамерау-Левенштейн)

```
1 func GetDistance(matr *matrix.Matrix, i int, j int, str1 []rune, str2 []rune) int {
2     if matr.Matr[i][j] == -1 {
3         matr.Matr[i][j] = RecursivePartOfLeven(str1, str2, matr)
4     }
5     return matr.Matr[i][j]
6 }
7
8 func RecursivePartOfLeven(str1 []rune, str2 []rune, matr *matrix.Matrix) int {
9     var operation int
10
11     m, n := len(str1), len(str2)
12
13     if m == 0 && n == 0 {
14         return 0
15     } else if m == 0 && n > 0 {
16         return n
17     } else if m > 0 && n == 0 {
18         return m
19     } else {
20         if str1[m-1] != str2[n-1] {
21             operation = 1
22         }
23     }
24 }
```

```

22     }
23     moveLeft := GetDistance(matr, m, n-1, str1, str2[:n-1]) + 1
24     moveRight := GetDistance(matr, m-1, n, str1[:m-1], str2) + 1
25     moveDiag := GetDistance(matr, m-1, n-1, str1[:m-1], str2[:n-1]) + operation
26     if m >= 2 && n >= 2 && str1[m-1] == str2[n-2] && str1[m-2] == str2[n-1] {
27         transposition := GetDistance(matr, m-2, n-2, str1[:m-2], str2[:n-2]) + 1
28         return GetMin(moveLeft, moveRight, moveDiag, transposition)
29     }
30     return GetMin(moveLeft, moveRight, moveDiag)
31 }
32 }
33
34 func RecursiveDamerauLevenWithCache(str1 []rune, str2 []rune) (int, matrix.Matrix) {
35     matr := matrix.CreateMatrix(len(str1)+1, len(str2)+1, true)
36     ans := RecursivePartOfLeven(str1, str2, &matr)
37     matr.Matr[matr.M-1][matr.N-1] = ans
38
39     return ans, matr
40 }

```

В листинге 3.5 приведён класс Matrix и его методы.

Листинг 3.5 – Класс Matrix

```

1 package matrix
2
3 import (
4     "fmt"
5 )
6
7 type Matrix struct {
8     Matr [][]int
9     M int
10    N int
11    Inf bool
12 }
13
14 func (matr *Matrix) MakeMatrix() {
15     matr.Matr = make([][]int, matr.M)
16     for i := range matr.Matr {
17         matr.Matr[i] = make([]int, matr.N)
18     }
19 }
20
21 func (matr *Matrix) InitMatrix() {
22     for j := 0; j < matr.N; j++ {
23         matr.Matr[0][j] = j
24     }
25 }

```

```

26     for i := 0; i < matr.M; i++ {
27         matr.Matr[i][0] = i
28     }
29
30     if matr.Inf == true {
31         for i := 1; i < matr.M; i++ {
32             for j := 1; j < matr.N; j++ {
33                 matr.Matr[i][j] = -1
34             }
35         }
36     }
37 }
38
39 func (matr *Matrix) OutputMatrix() {
40     for i := 0; i < matr.M; i++ {
41         for j := 0; j < matr.N; j++ {
42             fmt.Printf("%5d_", matr.Matr[i][j])
43         }
44         fmt.Println()
45     }
46     fmt.Println()
47 }
48
49 func CreateMatrix(m, n int, inf bool) Matrix {
50     matr := Matrix{M: m, N: n, Inf: inf}
51     matr.MakeMatrix()
52     matr.InitMatrix()
53     return matr
54 }

```

В листинге 3.6 приведен модуль для замеров процессорного времени выполнения программы.

Листинг 3.6 – Измерение времени

```

1 package time_measure
2
3 import (
4     "math/rand"
5     "syscall"
6
7     "lab1.com/algs1"
8     "lab1.com/matrix"
9 )
10
11 const N int = 1000
12 const symbols string = "abcdefghijklmnopqrstuvwxyz"
13 const MaxInd int = 26
14

```

```

15 func GetCPU() int64 {
16     usage := new(syscall.Rusage)
17     syscall.Getrusage(syscall.RUSAGE_SELF, usage)
18     return usage.Utime.Nano() + usage.Stime.Nano()
19 }
20
21 func GetRandomString(len int) []rune {
22     rstr := make([]rune, len)
23     for i := 0; i < len; i++ {
24         symb := rand.Intn(MaxInd)
25         rstr[i] = rune(symbols[symb])
26     }
27     return rstr
28 }
29
30 func MatrixLevenTimeMeasurement(str1 []rune, str2 []rune) (float32, int) {
31     var sum float32
32
33     var startTime, finishTime int64
34     var ans int
35     var mtr matrix.Matrix
36
37     for i := 0; i < N; i++ {
38         startTime = GetCPU()
39         ans, mtr = algs1.MatrixLeven(str1, str2)
40         finishTime = GetCPU()
41         sum += float32(finishTime - startTime)
42     }
43
44     mtr.Matr[0][0] += 1
45
46     return (sum / float32(N)) / 1e+9, ans
47 }
48
49 func DamerauLevenTimeMeasurement(str1 []rune, str2 []rune) (float32, int) {
50     var sum float32
51
52     var startTime, finishTime int64
53     var ans int
54
55     for i := 0; i < N; i++ {
56         startTime = GetCPU()
57         ans = algs1.DamerauLeven(str1, str2)
58         finishTime = GetCPU()
59         sum += float32(finishTime - startTime)
60     }
61
62     return (sum / float32(N)) / 1e+9, ans

```

```

63 }
64
65 func MatrixDamerauLevenTimeMeasurement(str1 []rune, str2 []rune) (float32, int) {
66     var sum float32
67
68     var startTime, finishTime int64
69     var ans int
70     var matr matrix.Matrix
71
72     for i := 0; i < N; i++ {
73         startTime = GetCPU()
74         ans, matr = algs1.MatrixDamerauLeven(str1, str2)
75         finishTime = GetCPU()
76         sum += float32(finishTime - startTime)
77     }
78
79     matr.Matr[0][0] += 1
80
81     return (sum / float32(N)) / 1e+9, ans
82 }
83
84 func RecursiveDamerauLevenWithCacheTimeMeasurement(str1 []rune, str2 []rune) (float32,
85     int) {
86     var sum float32
87
88     var startTime, finishTime int64
89     var ans int
90     var matr matrix.Matrix
91
92     for i := 0; i < N; i++ {
93         startTime = GetCPU()
94         ans, matr = algs1.RecursiveDamerauLevenWithCache(str1, str2)
95         finishTime = GetCPU()
96         sum += float32(finishTime - startTime)
97     }
98
99     matr.Matr[0][0] += 1
100
101     return (sum / float32(N)) / 1e+9, ans
102 }
103
104 func MeasureTime(rstr1 []rune, rstr2 []rune) [4]float32 {
105     var seconds [4]float32
106     var ans int
107
108     seconds[0], ans = MatrixLevenTimeMeasurement(rstr1, rstr2)
109     seconds[1], ans = DamerauLevenTimeMeasurement(rstr1, rstr2)
110     seconds[2], ans = MatrixDamerauLevenTimeMeasurement(rstr1, rstr2)

```

```

110     seconds[3], ans = RecursiveDamerauLevenWithCacheTimeMeasurement(rstr1, rstr2)
111
112     ans += 1
113
114     return seconds
115 }

```

3.4 Функциональное тестирование

В таблице 3.1 приведены функциональные тесты для алгоритмов вычисления расстояния Левенштейна и Дameraу-Левенштейна.

Обозначим:

- 1 - Левенштейн;
- 2 - Дameraу-Левенштейн;
- 3 - Дameraу-Левенштейн матричный;
- 4 - Дameraу-Левенштейн с кешем.

Таблица 3.1 – Функциональные тесты

Строка 1	Строка 2	Ожидаемый результат			
		1	2	3	4
equal	equal	0	0	0	0
call	code	2	2	2	2
program	prorgam	2	1	1	1
program	programmer	3	3	3	3

Вывод

В результате, были разработаны следующие алгоритмы: алгоритм нахождения редакционного расстояния Левенштейна, Дameraу-Левенштейна, алгоритм Дameraу-Левенштейна итерационный, алгоритм Дameraу-Левенштейна с кешированием.

4 Исследовательская часть

4.1 Пример работы

Демонстрация работы программы приведена на рисунке 4.1.

```
Выберите опцию: 1
Введите строку №1: one

Введите строку №2: two

Выберите опцию: 3

Редакционное расстояние между строками = 3
  0   1   2   3
  1   1   2   2
  2   2   2   3
  3   3   3   3

Выберите опцию: 4

Редакционное расстояние между строками = 3

Выберите опцию: 5

Редакционное расстояние между строками = 3
Матрица:
  0   1   2   3
  1   1   2   2
  2   2   2   3
  3   3   3   3

Выберите опцию: 6

Редакционное расстояние между строками = 3
Матрица:
  0   1   2   3
  1   1   2   2
  2   2   2   3
  3   3   3   3
```

Рисунок 4.1 – Демонстрация работы программы

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- операционная система: Ubuntu [3] 22.04.3 LTS;
- оперативная память: 15 GB;
- процессор: 12 * AMD Ryzen 5 4600H with Radeon Graphics [4].

4.3 Время выполнения алгоритмов

Во время замеров времени система не была нагружена дополнительными пользовательскими процессами.

Замеры проводились с помощью разработанной функции `GetCPU`, которая использует модуль `syscall` [5]. `syscall.Rusage` является структурой в пакете `syscall`, предназначенной для хранения информации о ресурсах, используемых процессом или потоком. Эта структура содержит различные поля, такие как `Utime` (время использования ЦПУ в пользовательском режиме), `Stime` (время использования ЦПУ в режиме ядра) и так далее.

Для получения времени использования ЦПУ, извлекаем атрибуты структуры `Utime` и `Stime`.

Выражение `usage.Utime.Nano() + usage.Stime.Nano()` возвращает сумму времени использования ЦПУ в наносекундах для процесса или потока. Это значение может быть использовано для измерения общего времени использования ЦПУ в пределах процесса или потока.

Результаты замеров приведены в таблице 4.1.

Обозначим:

- 1 - Матричный Левенштейн;
- 2 - Матричный Дамерау-Левенштейн;
- 3 - Рекурсивный Дамерау-Левенштейн;
- 4 - Рекурсивный Дамерау-Левенштейн с кешем.

Таблица 4.1 – Замер времени для строк, размером от 5 до 300

Длина строк	Время, с			
	1	2	3	4
5	2.34e-06	1.86e-06	2.88e-06	2.61e-06
10	5.17e-06	2.62e-06	6.38e-02	4.16e-06
15	9.31e-06	7.62e-06	2.1e+02	1.48e-05
50	6.31e-05	6.07e-05		1.53e-04
100	2.15e-04	2.32e-04		7.12e-04
200	8.76e-04	8.84e-04		2.66e-03
300	2.03e-03	2.06e-03		6.94e-03

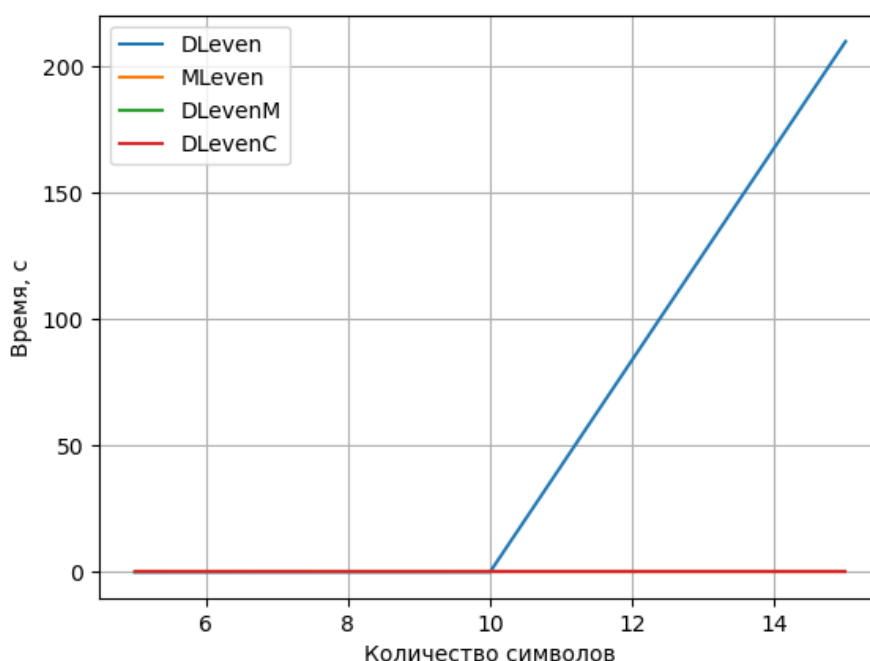


Рисунок 4.2 – Зависимость времени работы алгоритма вычисления расстояния Левенштейна и Дамерау-Левенштейна от длины строк (до 15 символов)

4.4 Использование памяти

Алгоритмы, использующие матрицы, не отличаются друг от друга по потреблению памяти, поэтому **сравним** только рекурсивную и матричную реализации.

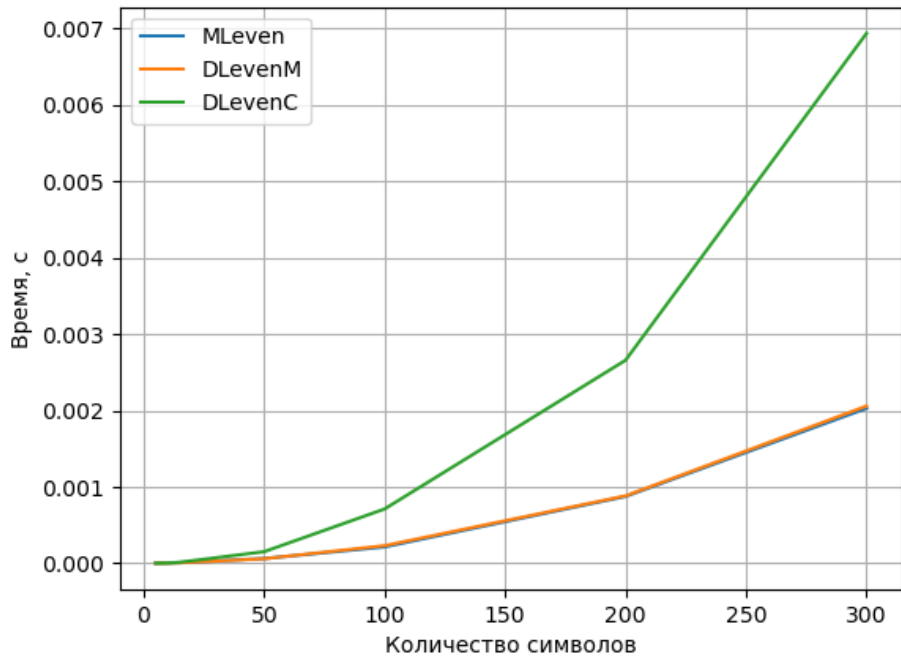


Рисунок 4.3 – Зависимость времени работы алгоритма вычисления расстояния Левенштейна и Дameraу-Левенштейна от длины строк (до 300 символов, без рекурсивной реализации алгоритма Дameraу-Левенштейна)

Матричный алгоритм Дameraу-Левенштейна:

$$11 \cdot \text{len}(\text{int}) + \text{len}(\text{rune}) \cdot (\text{len}(S_1) + \text{len}(S_2)) + \text{len}(\text{Matrix}), \quad (4.1)$$

где

$$\text{len}(\text{Matrix}) = (\text{len}(S_1) + \text{len}(S_2) + 2) \cdot \text{len}(\text{int}) + 2 \cdot \text{len}(\text{int}) + \text{len}(\text{bool}), \quad (4.2)$$

где len — оператор вычисления размера, S_1, S_2 — строки, `int` — целочисленный тип, `rune` — строковый тип в G`O`, `Matrix` — пользовательский тип данных, представляющий матрицу, `bool` — логический тип данных.

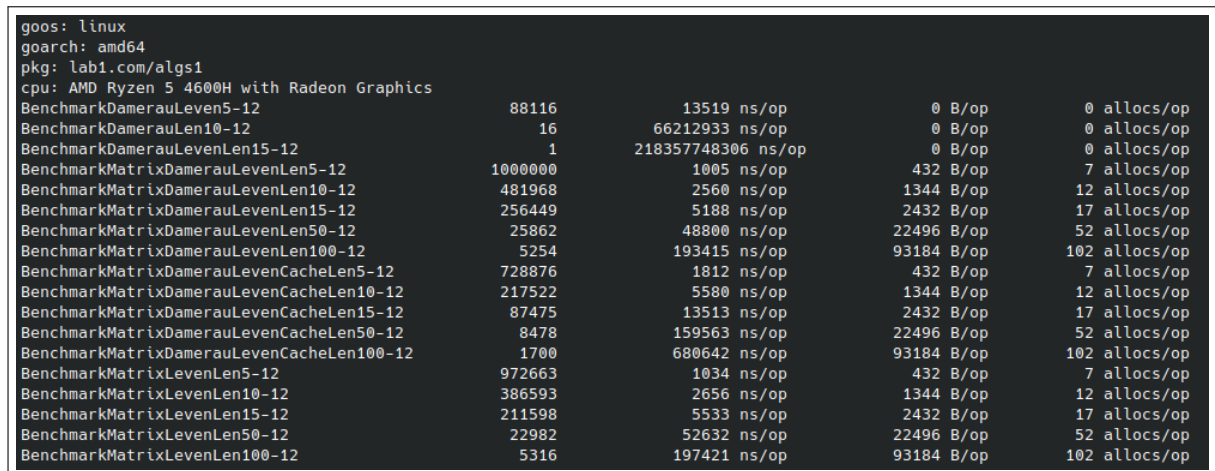
Рекурсивный алгоритм Дameraу-Левенштейна: Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк, соответственно, максимальный расход памяти

$$(\text{len}(S_1) + \text{len}(S_2)) \cdot (2 \cdot \text{len}(\text{rune}) + 15 \cdot \text{len}(\text{int})), \quad (4.3)$$

где len — оператор вычисления размера, S_1, S_2 — строки, `int` — целочис-

ленный тип, rune — строковый тип в GO.

Выделение памяти при работе алгоритмов указано на рисунке 4.4.



goos: linux				
goarch: amd64				
pkg: lab1.com/algs1				
cpu: AMD Ryzen 5 4600H with Radeon Graphics				
BenchmarkDamerauLeven5-12	88116	13519 ns/op	0 B/op	0 allocs/op
BenchmarkDamerauLeven10-12	16	66212933 ns/op	0 B/op	0 allocs/op
BenchmarkDamerauLeven15-12	1	218357748306 ns/op	0 B/op	0 allocs/op
BenchmarkMatrixDamerauLevenLen5-12	1000000	1005 ns/op	432 B/op	7 allocs/op
BenchmarkMatrixDamerauLevenLen10-12	481968	2560 ns/op	1344 B/op	12 allocs/op
BenchmarkMatrixDamerauLevenLen15-12	256449	5188 ns/op	2432 B/op	17 allocs/op
BenchmarkMatrixDamerauLevenLen50-12	25862	48800 ns/op	22496 B/op	52 allocs/op
BenchmarkMatrixDamerauLevenLen100-12	5254	193415 ns/op	93184 B/op	102 allocs/op
BenchmarkMatrixDamerauLevenCacheLen5-12	728876	1812 ns/op	432 B/op	7 allocs/op
BenchmarkMatrixDamerauLevenCacheLen10-12	217522	5580 ns/op	1344 B/op	12 allocs/op
BenchmarkMatrixDamerauLevenCacheLen15-12	87475	13513 ns/op	2432 B/op	17 allocs/op
BenchmarkMatrixDamerauLevenCacheLen50-12	8478	159563 ns/op	22496 B/op	52 allocs/op
BenchmarkMatrixDamerauLevenCacheLen100-12	1700	680642 ns/op	93184 B/op	102 allocs/op
BenchmarkMatrixLevenLen5-12	972663	1034 ns/op	432 B/op	7 allocs/op
BenchmarkMatrixLevenLen10-12	386593	2656 ns/op	1344 B/op	12 allocs/op
BenchmarkMatrixLevenLen15-12	211598	5533 ns/op	2432 B/op	17 allocs/op
BenchmarkMatrixLevenLen50-12	22982	52632 ns/op	22496 B/op	52 allocs/op
BenchmarkMatrixLevenLen100-12	5316	197421 ns/op	93184 B/op	102 allocs/op

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

Вывод

Рекурсивный алгоритм Дамерау-Левенштейна имеет большую вычислительную сложность и работает медленнее итеративных реализаций. Время выполнения рекурсивного алгоритма увеличивается экспоненциально с ростом длины строк. Например, на словах длиной 10 символов, матричная реализация алгоритма Дамерау-Левенштейна работает **в минимум** 12000 раз быстрее, чем рекурсивная реализация.

Однако, рекурсивный Дамерау-Левенштейна имеет преимущество в использовании памяти. Итеративные алгоритмы требуют большего объема памяти, так как их потребление памяти растет как произведение длин строк. В то же время, рекурсивный алгоритм требует памяти пропорционально сумме длин строк.

Алгоритм Дамерау-Левенштейна по времени выполнения сопоставим с алгоритмом Левенштейна.

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы были выполнены все поставленные задачи, в том числе изучение методов динамического программирования на основе алгоритмов вычисления расстояния Левенштейна.

Эксперименты позволили выявить различия в производительности различных алгоритмов вычисления расстояния Левенштейна. В частности, матричные алгоритмы продемонстрировали большее потребление памяти по сравнению с рекурсивными из-за выделения дополнительной памяти под матрицы и использования большего количества локальных переменных.

Также были проведены теоретические расчеты использования памяти в каждом из алгоритмов вычисления расстояния Левенштейна, включая алгоритм Дамерау-Левенштейна. В результате рекурсивный алгоритм продемонстрировал более эффективное использование памяти по сравнению с матричными алгоритмами, которые требуют дополнительного выделения памяти под матрицы и более широкий набор локальных переменных.

В целом, исследование позволило получить практические и теоретические результаты, подтверждающие различия в производительности и использовании памяти различных алгоритмов вычисления расстояния Левенштейна. Эти результаты могут служить основой для выбора наиболее эффективного алгоритма в зависимости от конкретных требований и ограничений проекта.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

- [1] Левенштейн В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов. – М.: Доклады АН СССР, 1965. Т. 163. С. 845–848.
- [2] The Go Programming Language [Электронный ресурс]. Режим доступа: <https://golang.org/> (дата обращения: 15.09.2023).
- [3] Linux – Википедия [Электронный ресурс]. Режим доступа: <https://ru.wikipedia.org/wiki/Ubuntu> (дата обращения: 15.09.2023).
- [4] AMD Processors [Электронный ресурс]. Режим доступа: <https://www.amd.com/en.html> (дата обращения: 15.09.2023).
- [5] Пакет syscall [Электронный ресурс]. Режим доступа: <https://pkg.go.dev/syscall> (дата обращения: 15.09.2023).