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 Расстояние Левенштейна и Дамерау-Левенштейна

Студент Богаченко А.Е.

Группа ИУ7-56Б

Оценка (баллы) _____

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

Москва — 2020 г.

Оглавление

Введение

Целью данной лабораторной работы является изучить и реализовать алгоритмы нахождения расстояний Левенштейна и Дамерау – Левенштейна.

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 лабораторной 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;

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 значе-

ния. 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

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

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.

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

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

Вывод

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

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.

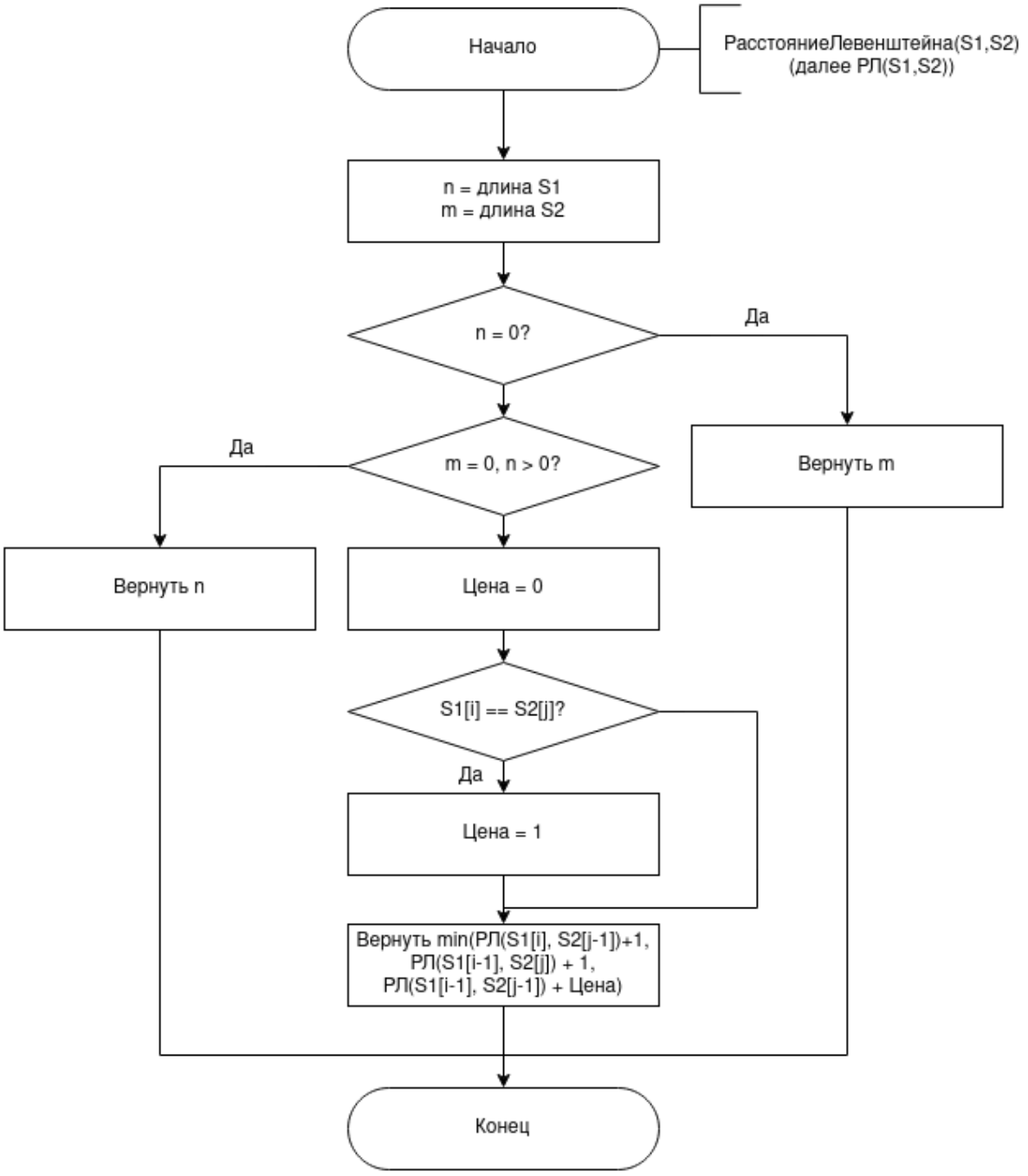

Рисунок 2.1 – Схема рекурсивного алгоритма Левенштейна

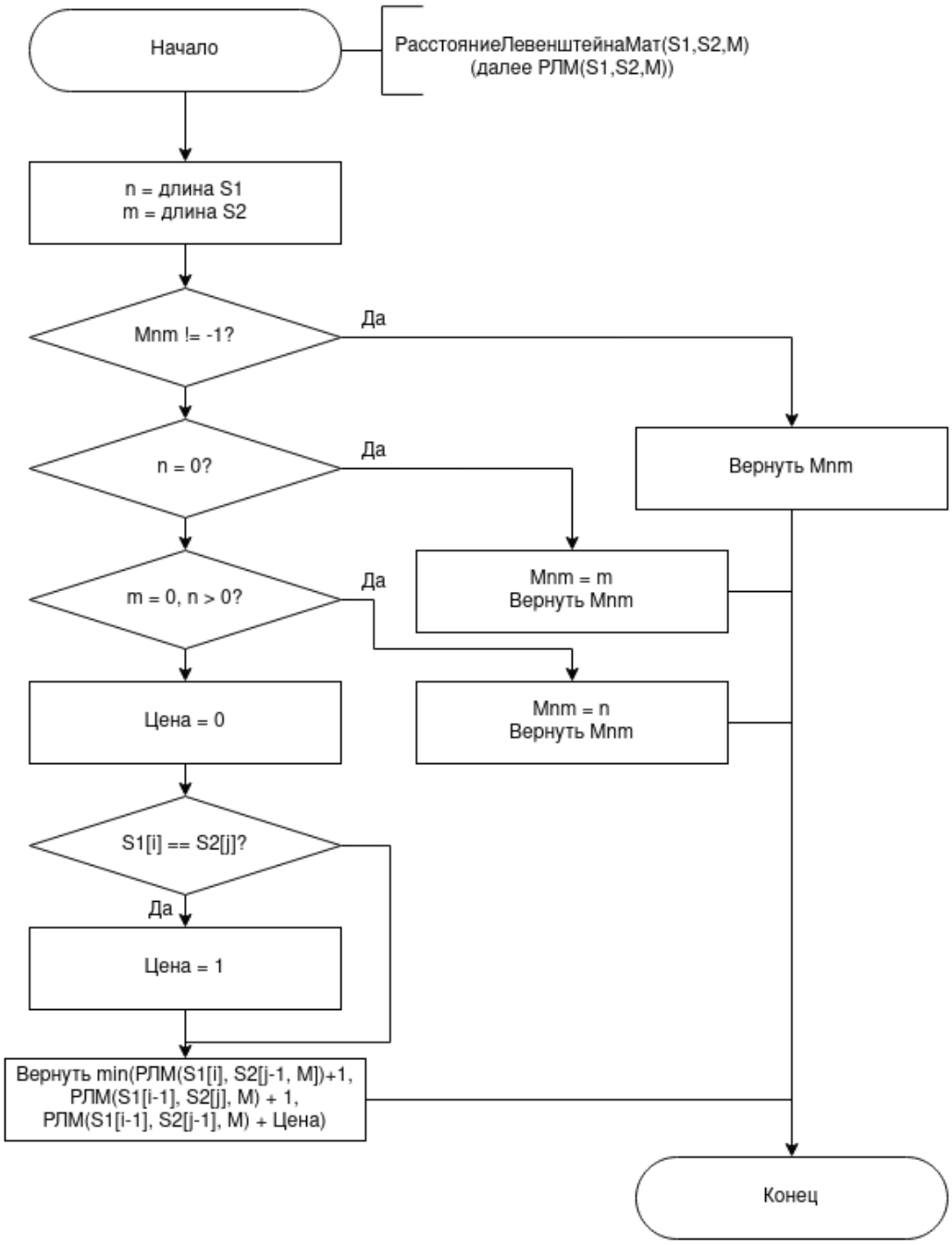

Рисунок 2.2 – Схема рекурсивного алгоритма Левенштейна с заполнением матрицы

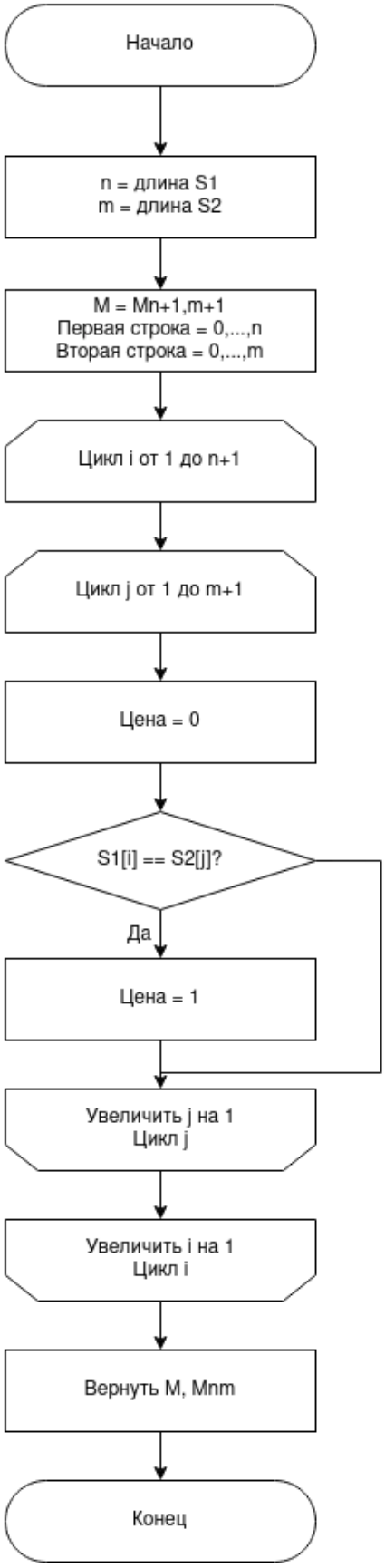

Рисунок 2.3 – Схема матричного алгоритма Левенштейна

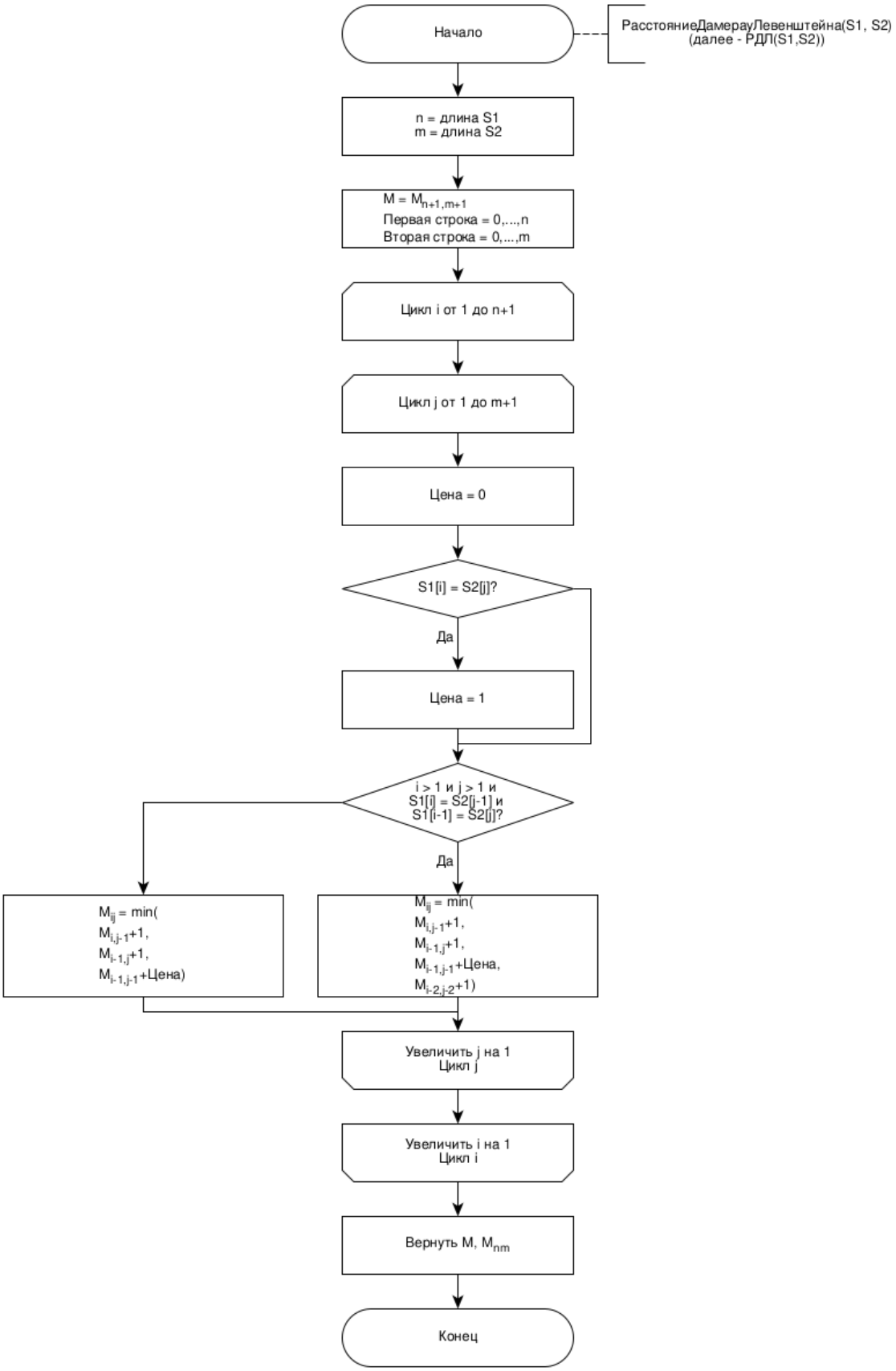

Рисунок 2.4 – Схема алгоритма Дамерау – Левенштейна

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 подаются две строки в любой языковой 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 GO [2]. Данный выбор обусловлен моим желанием расширить свои знания в области применения данного язкыа. Так же данный язык предоставляет средства тестирования разработанного ПО.

3.3 Листинг кода

В листингах ??–3.6 приведены реализации алгоритмов Левенштейна и Дамерау — Левенштейна, а также вспомогательные функции.

Листинг 3.1 – Рекурсивный

```
1 // LevRec - recursive levenstein
2 func LevRec(a, b string) int {
3     s1, s2 := []rune(a), []rune(b)
4 
5     lenS1, lenS2 := len(s1), len(s2)
6     // swap to save some memory O(min(a,b)) instead of O(a)
7     if lenS1 > lenS2 {
8         s1, s2 = s2, s1
```

```
        lenS1, lenS2 = lenS2, lenS1
    }

    return getDistRec(s1, s2, lenS1, lenS2)
}
```

Листинг 3.2 – Матричный

```
// LevMtr - levenstein with matrix
func LevMtr(a, b string) (d int, mtr [][]int) {
    s1, s2 := []rune(a), []rune(b)
    lenS1, lenS2 := len(s1)+1, len(s2)+1

    mtr = createMatrix(lenS1, lenS2, false)

    penalty := 0
    for i := 1; i < lenS1; i++ {
        for j := 1; j < lenS2; j++ {
            if s1[i-1] == s2[j-1] {
                penalty = 0
            } else {
                penalty = 1
            }

            mtr[i][j] = minOf(
                mtr[i-1][j]+1,
                mtr[i][j-1]+1,
                mtr[i-1][j-1]+penalty)
        }
    }
    d = mtr[lenS1-1][lenS2-1]

    return
}
```

Листинг 3.3 – Рекурсивный с заполнением матрицы

```
// LevMtrRec - Recursive Levenstein with matrix
func LevMtrRec(a, b string) (d int, mtr [][]int) {
    s1, s2 := []rune(a), []rune(b)
    lenS1, lenS2 := len(s1), len(s2)

    mtr = createMatrix(lenS1, lenS2, true)
    d = getDistMtr(s1, s2, lenS1, lenS2, mtr)

    return
}
```

Листинг 3.4 – Дамерау-Левенштейн

```
// DamLevMtr - Damerau - Levenshtein with matrix
func DamLevMtr(a, b string) (d int, mtr [][]int) {
    s1, s2 := []rune(a), []rune(b)
    lenS1, lenS2 := len(s1)+1, len(s2)+1

    mtr = createMatrix(lenS1, lenS2, false)

    penalty := 0
    for i := 1; i < lenS1; i++ {
        for j := 1; j < lenS2; j++ {
            if s1[i-1] == s2[j-1] {
                penalty = 0
            } else {
                penalty = 1
            }
            mtr[i][j] = minOf(
                mtr[i-1][j]+1,
                mtr[i][j-1]+1,
                mtr[i-1][j-1]+penalty)

            if i > 1 && j > 1 &&
                s1[i-1] == s2[j-2] &&
                s1[i-2] == s2[j-1] {
                mtr[i][j] = minOf(mtr[i][j], mtr[i-2][j-2]+1)
            }
        }
    }

    d = mtr[lenS1-1][lenS2-1]

    return
}
```

Листинг 3.5 – Вспомошательные функции

```
func getDistRec(a, b []rune, lenA, lenB int) int {
    if lenA == 0 {
        return lenB
    }
    if lenB == 0 && lenA > 0 {
        return lenA
    }
    penalty := 1
    if a[lenA-1] == b[lenB-1] {
        penalty = 0
    }

```

```
    return minOf(
        getDistRec(a, b, lenA, lenB-1)+1,
        getDistRec(a, b, lenA-1, lenB)+1,
        getDistRec(a, b, lenA-1, lenB-1)+penalty)
}

func getDistMtr(a, b []rune, lenA, lenB int, mtr [][]int) int {
    if lenA == 0 {
        return lenB
    }
    if lenB == 0 {
        return lenA
    }

    if mtr[lenA-1][lenB-1] != -1 {
        return mtr[lenA-1][lenB-1]
    }

    if a[lenA-1] == b[lenB-1] {
        mtr[lenA-1][lenB-1] = getDistMtr(a, b, lenA-1, lenB-1, mtr)
        return mtr[lenA-1][lenB-1]
    }

    mtr[lenA-1][lenB-1] = 1 + minOf(
        getDistMtr(a, b, lenA, lenB-1, mtr),
        getDistMtr(a, b, lenA-1, lenB, mtr),
        getDistMtr(a, b, lenA-1, lenB-1, mtr))

    return mtr[lenA-1][lenB-1]
}
```

Листинг 3.6 – Дополнительные утилиты

```
package levenshtein

import "fmt"

// OutMatrix prints matrix to stdout
func OutMatrix(mtr [][]int) {
    fmt.Println("Result␣matrix:")
    for i := range mtr {
        for j := range mtr[i] {
            fmt.Printf("%3d␣", mtr[i][j])
        }
        fmt.Printf("\n")
    }
}

// createMatrix is used to create and fill matrix for levenstein
```

```go
func createMatrix(n, m int, fill bool) (mtr [][]int) {
    mtr = make([][]int, n)
    for i := range mtr {
        mtr[i] = make([]int, m)
        mtr[i][0] = i
    }

    for j := 1; j < m; j++ {
        mtr[0][j] = j
    }

    if fill {
        for i := range mtr {
            for j := range mtr[i] {
                mtr[i][j] = -1
            }
        }
    }

    return
}

// minOf finds minimum from sequence of ints
func minOf(args ...int) (min int) {
    min = args[0]

    for _, val := range args {
        if min > val {
            min = val
        }
    }

    return
}

// GetLine reads line from stdin
func GetLine() string {
    var input string
    fmt.Scanln(&input)

    return input
}
```

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

Таблица 3.1 – Функциональные тесты

		Ожидаемый результат	
Строка 1	**Строка 2**	**Левенштейн**	**Дамерау — Левенштейн**
happy	happy	0	0
cook	cooker	2	2
mother	money	3	3
minute	moment	5	5
probelm	problem	2	1

Вывод

Были разработаны и протестированы спроектированные алгоритмы: 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.

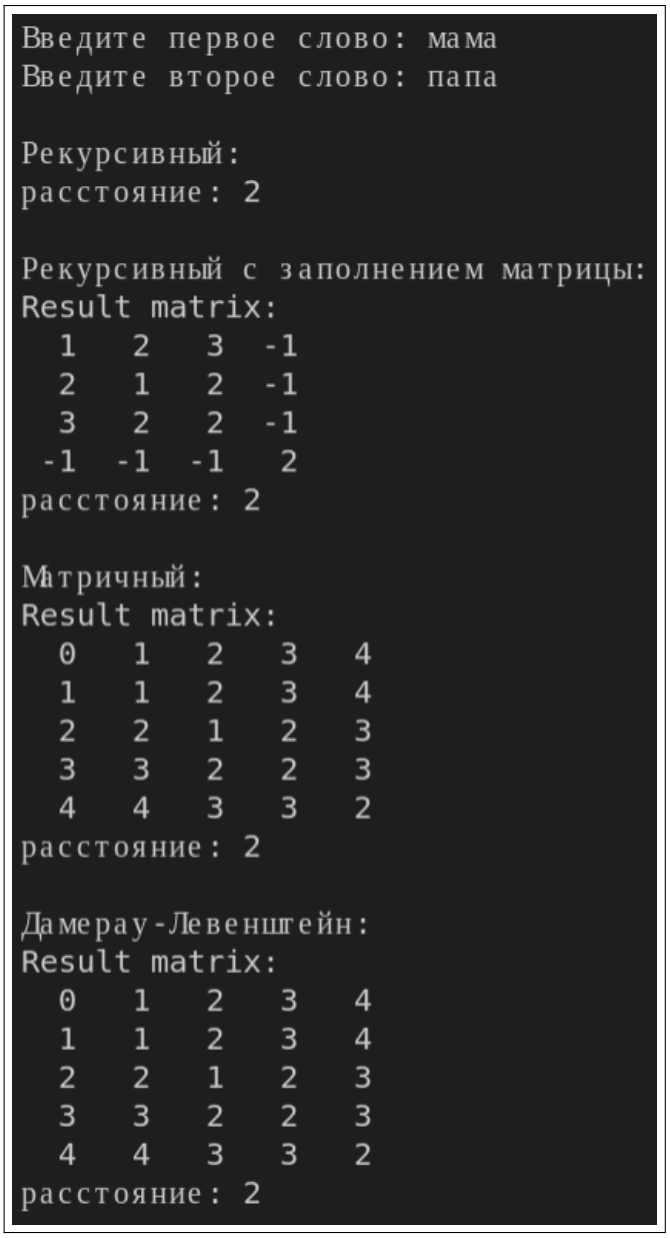

Рисунок 4.1 – Демонстрация работы алгоритмов нахождения расстояния Левенштейна и Дамерау – Левенштейна

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- Операционная система: Kali [3] Linux [4] 5.8.7-1kali1 64-bit.
- Память: 8 GB.
- Процессор: Intel® Core™ i5-8250U [5] CPU @ 1.60GHz

Тестирование проводилось на ноутбуке при включённом режиме производительности. Во время тестирования ноутбук был нагружен только системными процессами.

4.3 Время выполнения алгоритмов

Алгоритмы тестировались при помощи написания «бенчмарков» [6], предоставляемых 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.

Листинг 4.1 – Реализация бенчмарка

```
1  package levenshtein
2
3  import (
4      "testing"
5  )
6
7  // LevRec method benchmarks.
8
9  func BenchmarkRecursiveLen5(b *testing.B) {
10     s1 := "about"
11     s2 := "above"
12
13     for i := 0; i < b.N; i++ {
```

```
14          LevRec(s1, s2)
15      }
16  }
```

Результаты замеров приведены в таблице 4.1. В данной таблице для значений, для которых тестирование не выполнялось, в поле результата находится NaN. На рисунках 4.2 и 4.3 приведены графики зависимостей времени работы алгоритмов от длины строк.

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

Длина строк	**Время, нс**			
	LevRec	**RecMat**	**Mat**	**DamLev**
5	13182	NaN	NaN	NaN
10	63276060	2843	1499	1516
15	350648070708	NaN	NaN	NaN
20	NaN	10386	4444	5563
30	NaN	23108	9546	11715
50	NaN	60909	23567	28130
100	NaN	245804	100015	112921
200	NaN	1026719	468668	439941

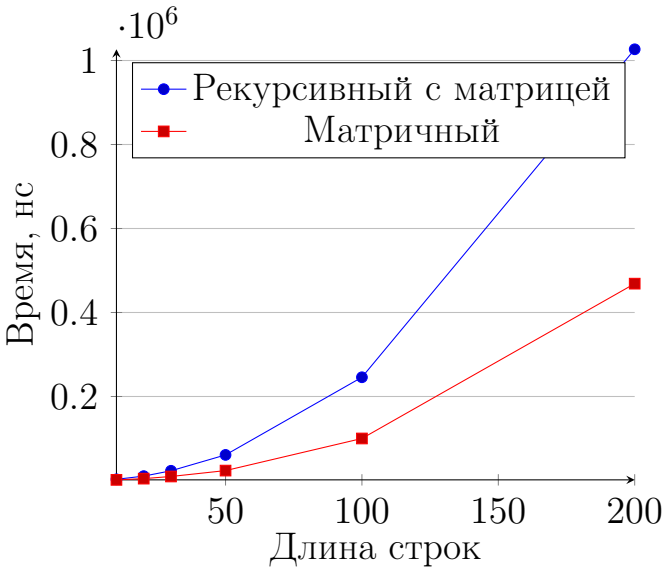

Рисунок 4.2 – Зависимость времени работы алгоритма вычисления расстояния Левенштейна от длины строк (рекурсивная с заполнением матрицы и матричная реализации)

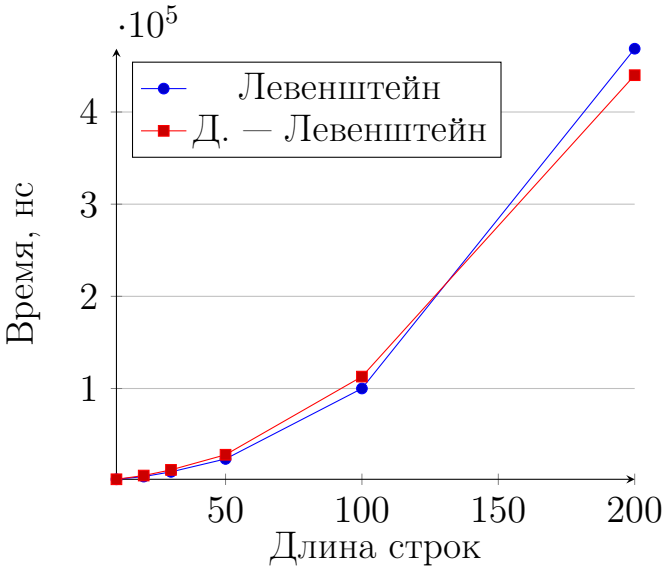

Рисунок 4.3 – Зависимость времени работы матричных реализаций алгоритмов Левенштейна и Дамерау — Левенштейна

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.

```
goos: linux
goarch: amd64
BenchmarkRecursiveLen5-8                    83779            13182 ns/op               0 B/op          0 allocs/op
BenchmarkRecursiveLen10-8                      16         63276060 ns/op               0 B/op          0 allocs/op
BenchmarkRecursiveLen15-8                       1        350648070708 ns/op            0 B/op          0 allocs/op
BenchmarkRecursiveMatrixLen10-8            427698             2843 ns/op            1344 B/op         12 allocs/op
BenchmarkRecursiveMatrixLen20-8            104208            10386 ns/op            4208 B/op         22 allocs/op
BenchmarkRecursiveMatrixLen30-8             51588            23108 ns/op            8704 B/op         32 allocs/op
BenchmarkRecursiveMatrixLen50-8             18198            60909 ns/op           22912 B/op         54 allocs/op
BenchmarkRecursiveMatrixLen100-8             4897           245804 ns/op           94016 B/op        104 allocs/op
BenchmarkRecursiveMatrixLen200-8              976          1026719 ns/op          366848 B/op        204 allocs/op
BenchmarkIterativeMatrixLen10-8            803342             1499 ns/op            1344 B/op         12 allocs/op
BenchmarkIterativeMatrixLen20-8            243945             4444 ns/op            4208 B/op         22 allocs/op
BenchmarkIterativeMatrixLen30-8            118819             9546 ns/op            8704 B/op         32 allocs/op
BenchmarkIterativeMatrixLen50-8             46011            23567 ns/op           22912 B/op         54 allocs/op
BenchmarkIterativeMatrixLen100-8            12330           100015 ns/op           94016 B/op        104 allocs/op
BenchmarkIterativeMatrixLen200-8             2527           468668 ns/op          366848 B/op        204 allocs/op
BenchmarkDamerauLevenshteinLen10-8         810336             1516 ns/op            1344 B/op         12 allocs/op
BenchmarkDamerauLevenshteinLen20-8         190770             5563 ns/op            4208 B/op         22 allocs/op
BenchmarkDamerauLevenshteinLen30-8         102396            11715 ns/op            8704 B/op         32 allocs/op
BenchmarkDamerauLevenshteinLen50-8          37952            28130 ns/op           22912 B/op         54 allocs/op
BenchmarkDamerauLevenshteinLen100-8          9727           112921 ns/op           94016 B/op        104 allocs/op
BenchmarkDamerauLevenshteinLen200-8          2371           439941 ns/op          366848 B/op        204 allocs/op
PASS
ok      _/root/bmstu-aa/lab1/src/levenshtein    380.154s
```

Рисунок 4.4 – Замеры производительности алгоритмов, выполненные при помощи команды `go test -bench . -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. Для слов длины 10 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. – М.: Доклады АН СССР, 1965. Т. 163. С. 845–848.

[2] The Go Programming Language [Электронный ресурс]. Режим доступа: `https://golang.org/` (дата обращения: 09.09.2020).

[3] Our Most Advanced Penetration Testing Distribution, Ever. [Электронный ресурс]. Режим доступа: `https://kali.org/` (дата обращения: 12.09.2020).

[4] Linux – Википедия [Электронный ресурс]. Режим доступа: `https://ru.wikipedia.org/wiki/Linux` (дата обращения: 12.09.2020).

[5] Intel Processors [Электронный ресурс]. Режим доступа: `https://www.intel.com/content/www/us/en/products/processors/core/i5-processors.html` (дата обращения: 12.09.2020).

[6] testing – The Go Programming Language [Электронный ресурс]. Режим доступа: `https://golang.org/pkg/testing/` (дата обращения: 12.09.2020).