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»

НАПРАВЛЕНИЕ ПОДГОТОВКИ «09.03.04 Программная инженерия»

ОТЧЕТ

ПО ЛАБОРАТОРНОЙ РАБОТЕ №1

Название: Расстояние Левенштейна и Дамерау – Левенштейна

Дисциплина: Анализ алгоритмов

Студент ИУ7-56Б ______________ Т. А. Казаева
Группа Подпись, дата И. О. Фамилия

Преподаватель ______________ Л. Л. Волкова
Подпись, дата И. О. Фамилия

Москва, 2021 г.

Оглавление

Страница

1 Введение **3**

2 Аналитический раздел **4**

2.1 Расстояние Левенштейна 4

2.2 Рекурсивная формула 4

2.3 Матрица расстояний 5

2.4 Рекурсивный алгоритм расстояния Левенштейна с мемоизацией 6

2.5 Расстояние Дамерау – Левенштейна 6

2.6 Вывод 7

3 Конструкторский раздел **8**

3.1 Матричные итерационные алгоритмы 8

3.2 Модификация матричных алгоритмов 8

3.3 Рекурсивные алгоритмы 8

3.4 Вывод 9

4 Технологический раздел **15**

4.1 Требования к ПО 15

4.2 Средства реализации 15

4.3 Листинги кода 15

Реализация алгоритмов 15

Утилиты 20

4.4 Тестовые данные 21

4.5 Вывод 22

5 Исследовательская часть **23**
5.1 Технические характеристики 23
5.2 Время выполнения алгоритмов 23
5.3 Использование памяти 25
5.4 Вывод 26

6 Заключение **27**

7 Список использованных источников **28**

1 Введение

Нахождение редакционного расстояния – одна из задач компьютерной лингвистики, которая находит применение в огромном количестве областей, начиная от предиктивных систем набора текста и заканчивая разработкой искусственного интеллекта. Впервые задачу поставил советский ученый В. И. Левенштейн [**Lev1965**], впоследствии её связали с его именем. В данной работе будут рассмотрены алгоритмы редакционного расстояния Левенштейна и расстояние Дамерау — Левенштейна.

Расстояния 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. Расстояние Дамерау — Левенштейна – модификация, добавляющая к редакторским операциям транспозицию, или обмен двух соседних символов местами.

Алгоритмы находят применение не только в компьютерной лингвистике (например, при реализации предиктивных систем при вводе текста), но и, например, при работе с утилитой diff и ей подобными. Также у алгоритма существуют более неочевидные применения, где операции проводятся не над буквами в естественном языке. Алгоритм применяется для распознавания текста на нечетких фотографиях. В этом случае сравниваются последовательности черных и белых пикселей на каждой строке изображения. Нередко алгоритм используется в биоинформатике для определения схожести разных участков ДНК или РНК.

Алгоритмы имеют некоторое количество модификаций, позволяющих эффективнее решать поставленную задачу. В данной работе будут предложены реализации алгоритмов, использующие парадигмы динамического программирования.

Цель лабораторной работы – получить навыки динамического программирования. Задачами лабораторной работы являются изучение и реализация алгоритмов Левенштейна и Дамерау — Левенштейна, применение парадигм динамического программирования при реализации алгоритмов и сравнительный анализ алгоритмов на основе экспериментальных данных.

2 | Аналитический раздел

2.1 Расстояние Левенштейна

Редакторское расстояние (расстояние Левенштейна) – это минимальное количество операций вставки, удаления и замены, необходимых для превращения одной строки в другую. Каждая редакторская операция имеет цену (штраф). В общем случае, имея на входе строку $X = x_1x_2...x_n$ и $Y = y_1y_2...y_n$, расстояние между ними можно вычислить с помощью операций:

- $\mathrm{delete}(u, \varepsilon) = \delta$
- $\mathrm{insert}(\varepsilon, v) = \delta$
- $\mathrm{replace}(u, v) = \alpha(u, v) \leq 0$ (здесь, $\alpha(u, u) = 0$ для всех u).

Необходимо найти последовательность замен с минимальным суммарным штрафом. Далее, цена вставки и удаления будет считаться равной 1.

2.2 Рекурсивная формула

Используя условные обозначения, описанные в разделе 2.1, рекурсивная формула для нахождения расстояния Левенштейна $f(i, j)$ между подстроками $x_1...x_i$ и $y_1...y_j$ имеет следующий вид:

$$f_{X,Y}(i,j) = \begin{cases} \delta_i & j = 0 \\ \delta_j & i = 0 \\ min \begin{cases} \alpha(x_i, y_i) + f_{X,Y}(i-1, j-1) \\ \delta + f_{X,Y}(i-1, j) \\ \delta + f_{X,Y}(i, j-1) \end{cases} & \text{иначе.} \end{cases} \tag{2.1}$$

$f_{X,Y}$ – редакционное расстояние между двумя подстроками – первыми i символами строки X и первыми j символами строки Y. Очевидны следующие утверждения:

- Если редакционное расстояние нулевое, то строки равны:
  $f_{X,Y} = 0 \Rightarrow X = Y$
- Редакционное расстояние симметрично:
  $f_{X,Y} = f_{Y,X}$
- Максимальное значение $f_{X,Y}$ – размерность более длинной строки:
  $f_{X,Y} \leq max(|X|, |Y|)$
- Минимальное значение $f_{X,Y}$ – разность длин обрабатываемых строк:
  $f_{X,Y} \geq abs(|X| - |Y|)$
- Аналогично свойству треугольника, редакционное расстояние между двумя строками не может быть больше чем редакционные расстояния каждой из этих строк с третьей:
  $f_{X,Y} \leq f_{X,Z} + f_{Z,Y}$

2.3 Матрица расстояний

В 2001 году был предложен подход, использующий динамическое программирование. Этот алгоритм, несмотря на низкую эффективность, один из самых гибких и может быть изменен в соответствии с функцией нахождения расстояния, по которой производится расчет[**Navarro2001**].

Пусть $C_{0..|X|,0..|Y|}$ – матрица расстояний, где $C_{i,j}$ – минимальное количество редакторских операций, необходимое для преобразования подстроки $x_1...x_i$ в подстроку $y_1...y_j$. Матрица заполняется следующим образом:

$$Ci, j = \begin{cases} i & j = 0 \\ j & i = 0 \\ min \begin{cases} C_{i-1,j-1} + \alpha(x_i, y_i), \\ C_{i-1,j} + 1, \\ C_{i,j-1} + 1) \end{cases} & \text{иначе.} \end{cases} \tag{2.2}$$

При решении данной задачи используется ключевая идея динамического программирования – чтобы решить поставленную задачу, требуется решить отдельные части задачи (подзадачи), после чего объединить решения подзадач в одно общее решение. Здесь небольшие подзадачи – это заполнение ячеек таблицы с индексами $i < |X|, j < |Y|$. После заполнения всех ячеек матрицы в ячейке $C_{|X|,|Y|}$ будет записано искомое расстояние.

2.4 Рекурсивный алгоритм расстояния Левенштейна с мемоизацией

При реализации рекурсивного алгоритма используется мемоизация – сохранение результатов выполнения функций для предотвращения повторных вычислений. Отличие от формулы 2.2 состоит лишь в начальной инициализации матрицы флагом ∞, котрый сигнализирует о том, была ли обработана ячейка. В случае если ячейка была обработана, алгоритм переходит к следующему шагу.

2.5 Расстояние Дамерау – Левенштейна

Расстояние Дамерау – Левенштейна – модификация расстояния Левенштейна, добавляющая транспозицию к редакторским операциям, предложенными Левенштейном (см. 2.1). изначально алгоритм разрабатывался для сравнения текстов, набранных человеком (Дамерау показал[**damerau**], что 80% человеческих ошибок при наборе текстов составляют перестановки соседних символов, пропуск символа, добавление нового символа, и ошибка в символе. Поэтому метрика Дамерау-Левенштейна часто используется в редакторских программах для проверки правописания).

Используя условные обозначения, описанные в разделе 2.1, рекурсивная формула для нахождения расстояния Дамерау – Левенштейна $f(i, j)$

между подстроками $x_1...x_i$ и $y_1...y_j$ имеет следующий вид:

$$f_{X,Y}(i,j) = \begin{cases} \delta_i & j = 0 \\ \delta_j & i = 0 \\ min \begin{cases} \alpha(x_i, y_i) + f_{X,Y}(i-1, j-1) \\ \delta + f_{X,Y}(i-1, j) \\ \delta + f_{X,Y}(i, j-1) \\ \left[\begin{array}{ll} \delta + f_{X,Y}(i-2, j-2) & i, j > 1 x_i = y_{j-1} x_{i-1} = y_j \\ \infty & \text{иначе;} \end{array}\right. \end{cases} & \text{иначе.} \end{cases} \tag{2.3}$$

2.6 Вывод

Обе вариации алгоритма редакторского расстояния могут быть реализованы как рекурсивно, так и итеративно. Итеративная реализация может быть осуществлена с помощью парадигм динамического программирования, используя матрицу расстояний. [**damerau**]

3 | Конструкторский раздел

В данном разделе представлены схемы реализуемых алгоритмов и их модификации.

3.1 Матричные итерационные алгоритмы

На рисунке 3.1 изображена схема алгоритма нахождения расстояния Левенштейна итерационно с использованием матрицы расстояний. На рисунке 3.4 изображена схема алгоритма нахождения расстояния Дамерау – Левенштейна итеративно с использованием матрицы расстояний.

3.2 Модификация матричных алгоритмов

Мемоизация - это прием сохранения промежуточных результатов, которые могут еще раз понадобиться в ближайшее время, чтобы избежать их повторного вычисления. Матричный алгоритм нахождения расстояния Левенштейна может быть модифицирован, используя мемоизацию – достаточно инициализировать матрицу значением ∞, которое будет рассмотрено в качестве флага. На рисунке 3.2 изображена схема алгоритма, использующая этот прием.

3.3 Рекурсивные алгоритмы

На рисунке 3.3 изображена схема рекурсивного алгоритма нахождения расстояния Левенштейна. На рисунке 3.5 изображена схема рекурсивного алгоритма нахождения расстояния Дамерау – Левенштейна.

3.4 Вывод

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

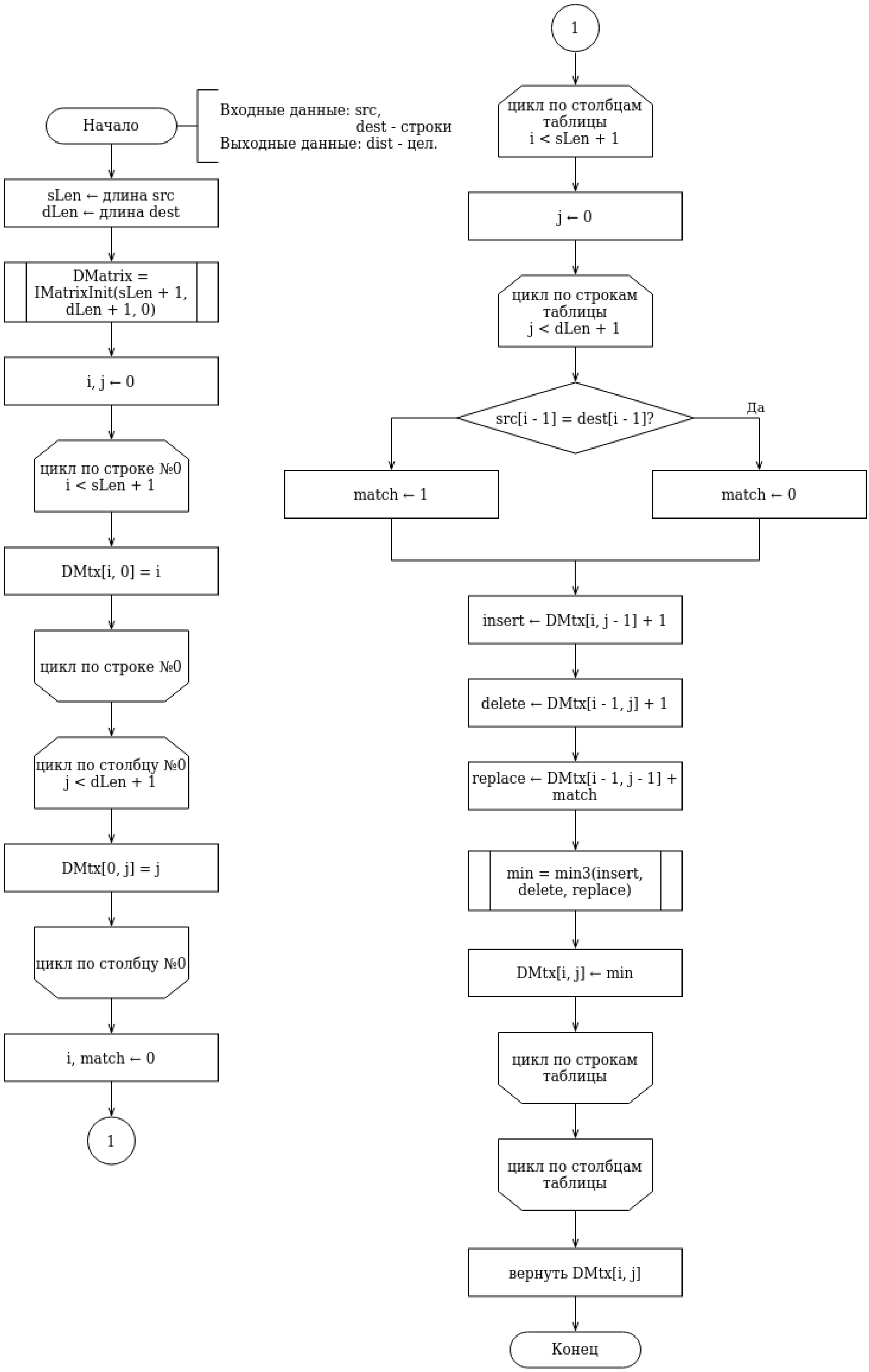

Рисунок 3.1 – Схема итерационного алгоритма расстояния Левенштейна с заполнением матрицы расстояний

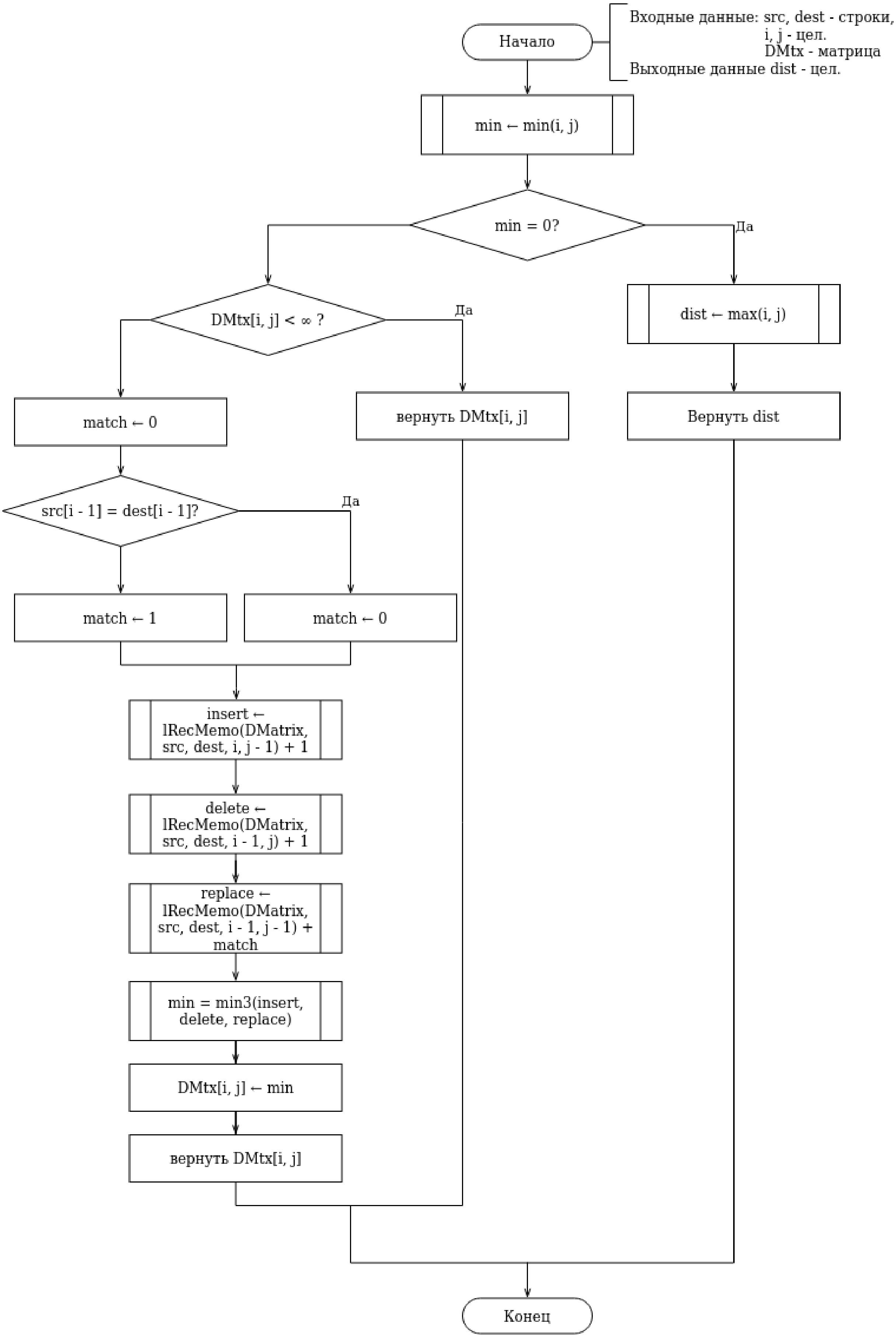

Рисунок 3.2 – Схема рекурсивного алгоритма расстояния Левенштейна с мемоизацией

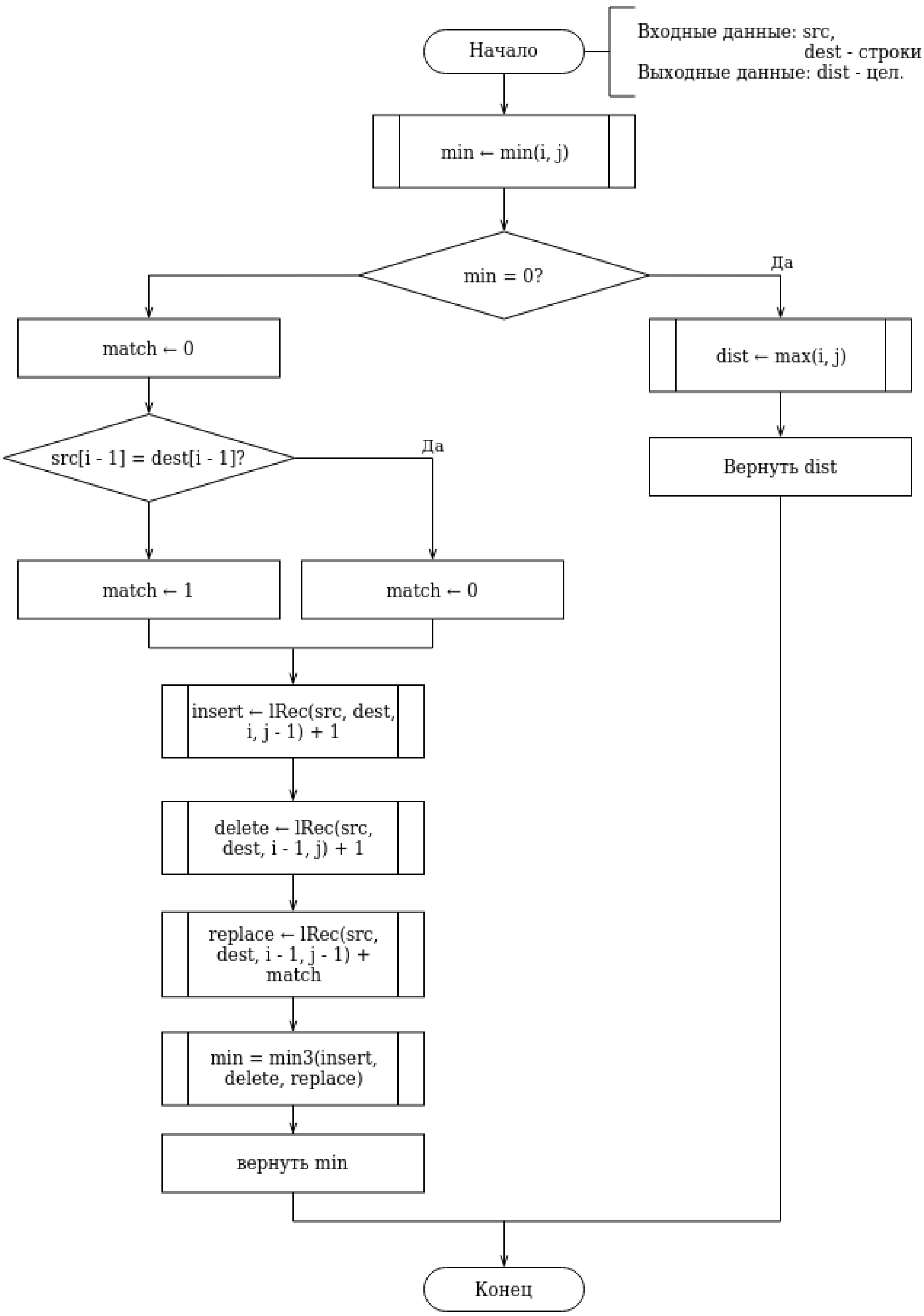

Рисунок 3.3 – Схема рекурсивного алгоритма расстояния Левенштейна

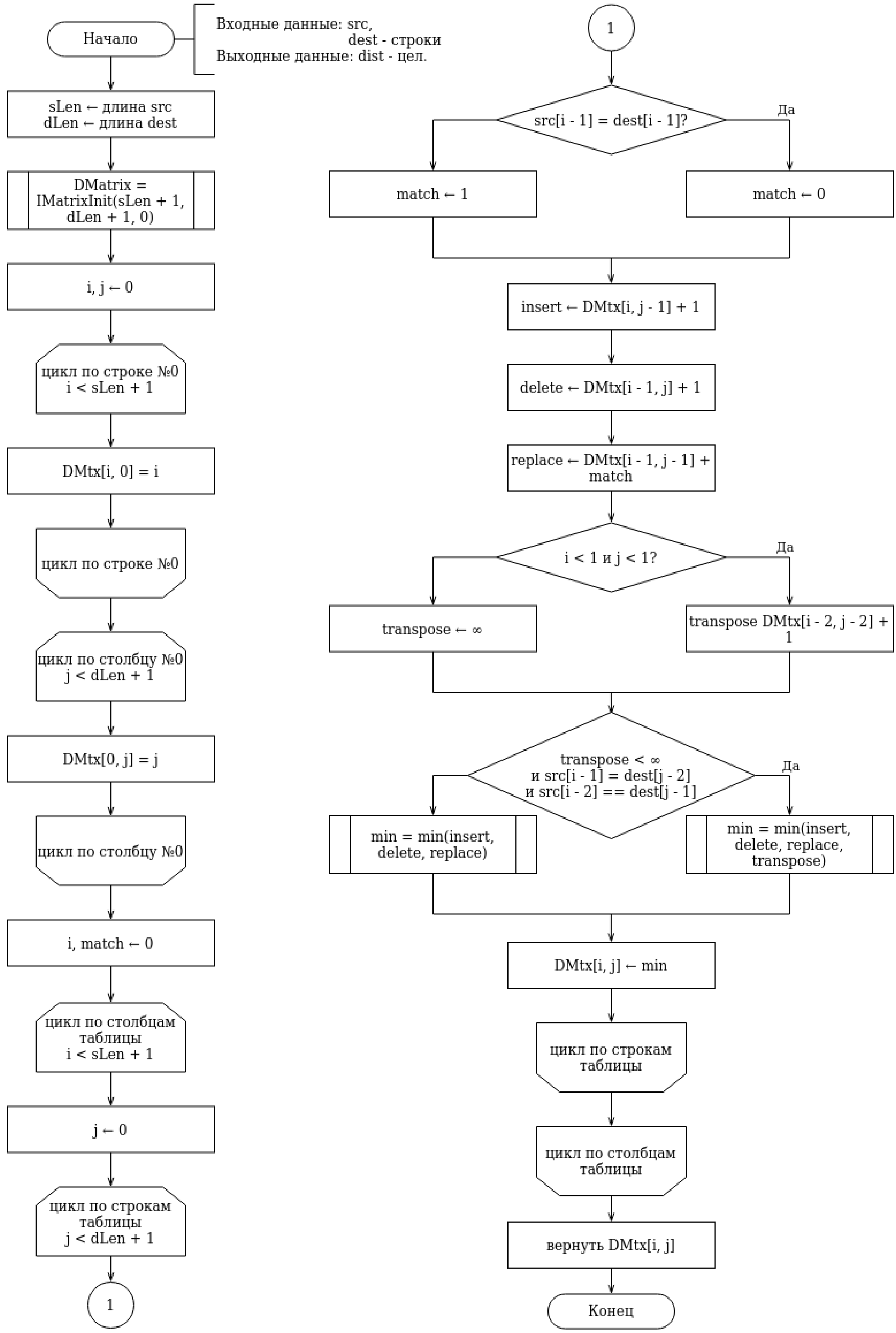

Рисунок 3.4 – Схема итерационного алгоритма расстояния Дамерау - Левенштейна с заполнением матрицы расстояний

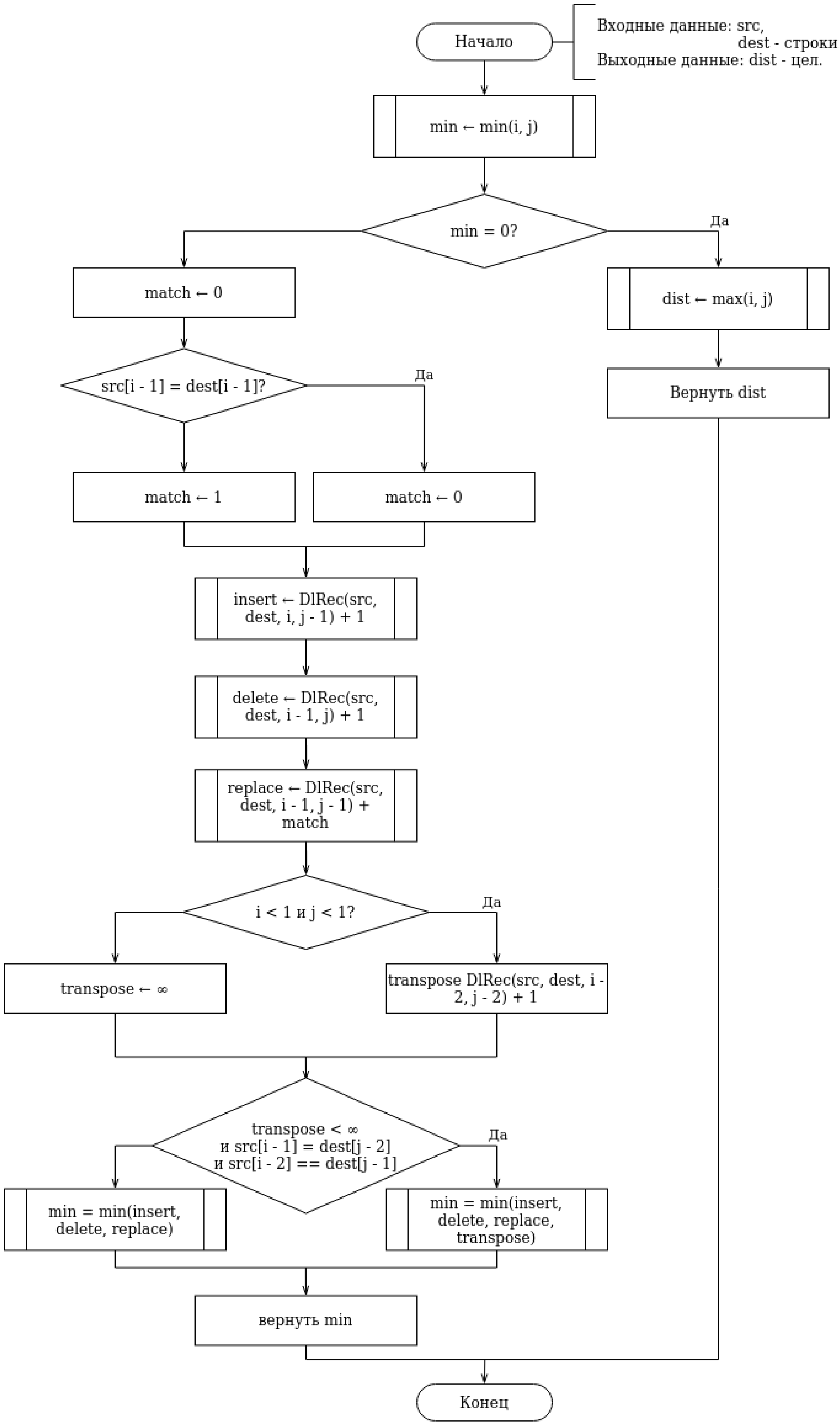

Рисунок 3.5 – Схема рекурсивного алгоритма расстояния Дамерау - Левенштейна

4 Технологический раздел

4.1 Требования к ПО

Программа должна отвечать следующим требованиям:

- ПО корректно реагирует на любые действия пользователя;
- ПО возвращает полученное расстояние;
- ПО принимает текстовые данные в любой раскладке.
- Время отклика программы на любое действие пользователя должно быть приемлемым.

4.2 Средства реализации

Для реализации ПО был выбран компилируемый многопоточный язык программирования Golang, поскольку язык отличается легкой и быстрой сборкой программ, автоматическим управлением памяти и понятным синтаксисом. В качестве среды разработки была выбрана среда VS Code, написание сценариев осуществлялось утилитой make.

4.3 Листинги кода

Реализация алгоритмов

В качестве представления строковых данных был выбран тип rune[1] – псевдоним для типа int32.

В листингах 4.1 - 4.5 приведены реализации алгоритмов, описанных в разделе 2.

Листинг 4.1 – Программный код нахождения расстояния Левенштейна итеративно

```
func LevenshteinMatrixIterative(src, dest []rune) int {
    srcLen, destLen := len(src), len(dest)

    rows, cols := srcLen + 1, destLen + 1
    DMtx := matrix.IMtxInit(rows, cols, 0)

    for i := 1; i < rows; i++ {
        DMtx[i][0] = i
    }
    for j := 1; j < cols; j++ {
        DMtx[0][j] = j
    }

    match := 0
    for i := 1; i < rows; i++ {
        for j := 1; j < cols; j++ {
            if src[i - 1] == dest[j - 1] {
                match = 0
            } else {
                match = 1
            }
            insert  := DMtx[i][j - 1] + 1
            delete  := DMtx[i - 1][j] + 1
            replace := DMtx[i - 1][j - 1] + match

            min := utils.MinOfInt(insert, delete, replace)
            DMtx[i][j] = min
        }
    }
    return DMtx[rows - 1][cols - 1]
}
```

В рекурсивной части алгоритма нахождения расстояния Левенштейна с мемоизацией в качестве входных данных выступают не только строки, но еще и матрица расстояний. Поскольку все функции нахождения расстояния приведены к единому виду, было решено разделить реализацию алгоритма на две части – часть с запуском рекурсии и начальной инициализацией и рекурсивная часть алгоритма. Таким образом, пользователь с матрицей расстояний никак не взаимодействует.

Листинг 4.2 – Программный код нахождения расстояния Левенштейна с мемоизацией

```
func LevenshteinPartialMemo(src, dest []rune) int {
    srcLen, destLen := len(src), len(dest)

    rows, cols := srcLen + 1, destLen + 1
    DMtx := matrix.IMtxInit(rows, cols, -1)

    ans := _partialMemoHelper(src, dest, srcLen, destLen, DMtx)
    return ans
}

func _partialMemoHelper(src, dest []rune, i, j int,
DMtx matrix.IMtx) int {
    if j == 0 {
        return i
    }
    if i == 0 {
        return j
    }

    if DMtx[i][j] != -1 {
        return DMtx[i][j]
    }

    match := 1
    if src[i - 1] == dest[j - 1] {
        match = 0
    }

    insert  := _partialMemoHelper(src, dest, i, j - 1, DMtx) + 1
    delete  := _partialMemoHelper(src, dest, i - 1, j, DMtx) + 1
    replace := match + _partialMemoHelper(src, dest, i - 1, j - 1, DMtx)

    min := utils.MinOfInt(insert, delete, replace)
    DMtx[i][j] = min

    return DMtx[i][j]
}
```

Аналогично листингу 4.2, реализация рекурсивного алгоритма разделена на две функции с целью унификации всех функций расстояния.

Листинг 4.3 – Программный код нахождения расстояния Левенштейна с рекурсией

```
func LevenshteinRecursive(src, dest []rune) int {
    srcLen, destLen := len(src), len(dest)

    ans := _lRecursiveHelper(src, dest, srcLen, destLen)

    return ans
}

func _lRecursiveHelper(src, dest []rune, i, j int) int {
    if (utils.MinOfInt(i, j) == 0) {
        return utils.Max2Int(i, j)
    }

    match := 1
    if (src[i - 1] == dest[j - 1]) {
        match = 0
    }

    insert := _lRecursiveHelper(src, dest, i, j - 1) + 1
    delete := _lRecursiveHelper(src, dest, i - 1, j) + 1
    replace := match + _lRecursiveHelper(src, dest, i - 1, j - 1)

    min := utils.MinOfInt(insert, delete, replace)
    return min
}
```

Листинг 4.4 – Программный код нахождения расстояния Дамерау – Левенштейна итеративно

```
func DamerauLevenshteinIterative(src, dest []rune) int{
    srcLen, destLen := len(src), len(dest)

    rows, cols := srcLen + 1, destLen + 1
    DMtx := matrix.IMtxInit(rows, cols, 0)

    for i := 1; i < rows; i++ {
        DMtx[i][0] = i
    }
    for j := 1; j < cols; j++ {
        DMtx[0][j] = j
    }

    match := 0
    min := 0
    for i := 1; i < rows; i++ {
        for j := 1; j < cols; j++ {
            if src[i - 1] == dest[j - 1] {
                match = 0
```

```
            } else {
                match = 1
            }

            insert := DMtx[i][j - 1] + 1
            delete := DMtx[i - 1][j] + 1
            replace := DMtx[i - 1][j - 1] + match
            transpose := -1

            if i > 1 && j > 1 {
                transpose = DMtx[i - 2][j - 2] + 1
            }

            if i > 1 && j > 1 && src[i - 1] == dest[j - 2]
            && src[i - 2] == dest[j - 1] {
                min = utils.MinOfInt(insert, delete, replace, transpose)
            } else {
                min = utils.MinOfInt(insert, delete, replace)
            }
            DMtx[i][j] = min
        }
    }
    DMtx[srcLen][destLen] = min
    return DMtx[srcLen][destLen]
}
```

Листинг 4.5 – Программный код нахождения расстояния Дамерау – Левенштейна рекурсивно

```
func DamerauLevenshteinRecursive(src, dest []rune) int {
    srcLen, destLen := len(src), len(dest)

    ans := _dRecursiveHelper(src, dest, srcLen, destLen)

    return ans
}

func _dRecursiveHelper(src, dest []rune, i, j int) int {
    if (utils.MinOfInt(i, j) == 0) {
        return utils.Max2Int(i, j)
    }

    match := 1
    if (src[i - 1] == dest[j - 1]) {
        match = 0
    }

    insert := _dRecursiveHelper(src, dest, i, j - 1) + 1
```

```
    delete := _dRecursiveHelper(src, dest, i - 1, j) + 1
    replace := match + _dRecursiveHelper(src, dest, i - 1, j - 1)

    transpose := -1

    if i > 1 && j > 1 {
        transpose =_dRecursiveHelper(src, dest, i - 2, j - 2) + 1
    }

    min := 0
    if transpose != -1 && src[i - 1] == dest[j - 2]
    && src[i - 2] == dest[j - 1] {
        min = utils.MinOfInt(insert, delete, replace, transpose)
    } else {
        min = utils.MinOfInt(insert, delete, replace)
    }
    return min
}
```

Утилиты

В листингах 4.6 - 4.8 приведены используемые утилиты.

Листинг 4.6 – Функция нахождения минимума из N целых чисел

```
func MinOfInt(values ...int) int {
    min := values[0]

    for _, i := range values {
        if min > i {
            min = i
        }
    }
    return min
}
```

Листинг 4.7 – Функция нахождения максимума из двух целых чисел

```
func Max2Int(v1, v2 int) int {
    if v1 < v2 {
        return v2
    }
    return v1
}
```

Листинг 4.8 – Определение типа целочисленной матрицы; его инициализация и вывод

```
type IMtx [][]int

func IMtxInit(rows, cols, filler int) IMtx {
    mtx := make(IMtx, rows)
    for i := range mtx {
        mtx[i] = make([]int, cols)
        for j := 0; j < cols; j++ {
            mtx[i][j] = filler
        }
    }
    return mtx
}

func IMtxLog(mtx IMtx, rows, cols int) {
    for i := 0; i < rows; i++ {
        for j := 0; j < cols; j++ {
            fmt.Print(mtx[i][j])
            fmt.Print(" ")
        }
        fmt.Println("")
    }
}
```

4.4 Тестовые данные

№	S_1	S_2	LIter	LRec	LRecMem	DLIter	DLRec
1	« »	« »	0	0	0	0	0
2	«book»	«back»	2	2	2	2	2
3	«critical»	«colleague»	8	8	8	8	8
4	«reptile»	«perfume»	2	2	2	2	2
5	«note»	«fset»	4	4	4	3	3
6	«bow»	«elbow»	2	2	2	2	2
6	«same»	«same»	0	0	0	0	0

4.5 Вывод

На основе схем из конструкторского раздела были разработаны программные реализации требуемых алгоритмов.

5 | Исследовательская часть

5.1 Технические характеристики

Тестирование выполнялось на устройстве со следующими техническими характеристиками:

- Операционная система Ubuntu 20.04.1 LTS;
- Память 7 GiB
- Процессор Intel(R) Core(TM) i3-8145U CPU @ 2.10GHz

Во время тестирования устройство было подключено к блоку питания и не нагружено никакими приложениями, кроме встроенных приложений окружения, окружением и системой тестирования.

5.2 Время выполнения алгоритмов

Алгоритмы тестировались при помощи профилирования – сбора характеристик работы программы: времени выполнения и затрат по памяти. Для каждой функции были написаны "бенчмарки"[2], представленные встроенными в Golang средствами. Бенчмарки, реализованные стандартными средствами Golang автоматически делают некоторое количество замеров, предоставляя результат с некоторой погрешностью.

Листинг 5.1 – Пример бенчмарка

```
func BenchmarkDamerauIterative20(b *testing.B) {
    src := []rune("zyapjbnxuykcwtnmexid")
    dest := []rune("ickfleanmgcpakskibcx")

    for i := 0; i < b.N; i++ {
        dist.DamerauLevenshteinIterative(src, dest)
    }
}
```

Результаты тестирования приведены в таблице .Прочерк в таблице означает что тестирование для этого набора данных не выполнялось.

Таблица 5.1 – Время выполнения алгоритмов

Длина строк	Время выполнения()				
	LIter	LRec	LRecMem	DLIter	DLRec
5	468	18762	959	531	22974
10	1261	91229882	3442	1519	139913520
15	2614	502162161429	7243	3144	981845877565
20	4244	-	12883	5309	-
30	8806	-	28096	11257	-
50	22944	-	76094	29965	-
100	88145	-	304941	117059	-
200	338736	-	1267569	451253	-

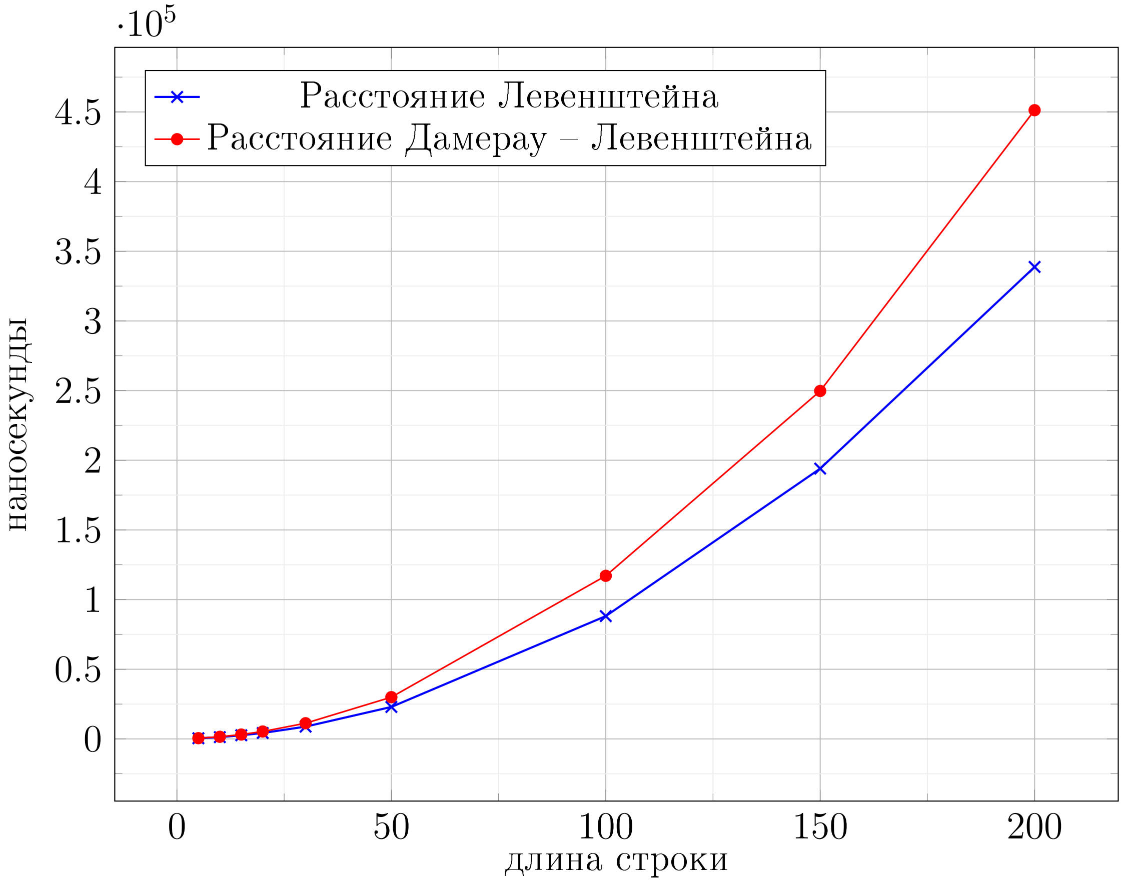

Рисунок 5.1 – Сравнение расстояния итеративного и матричного расстояния Левенштейна

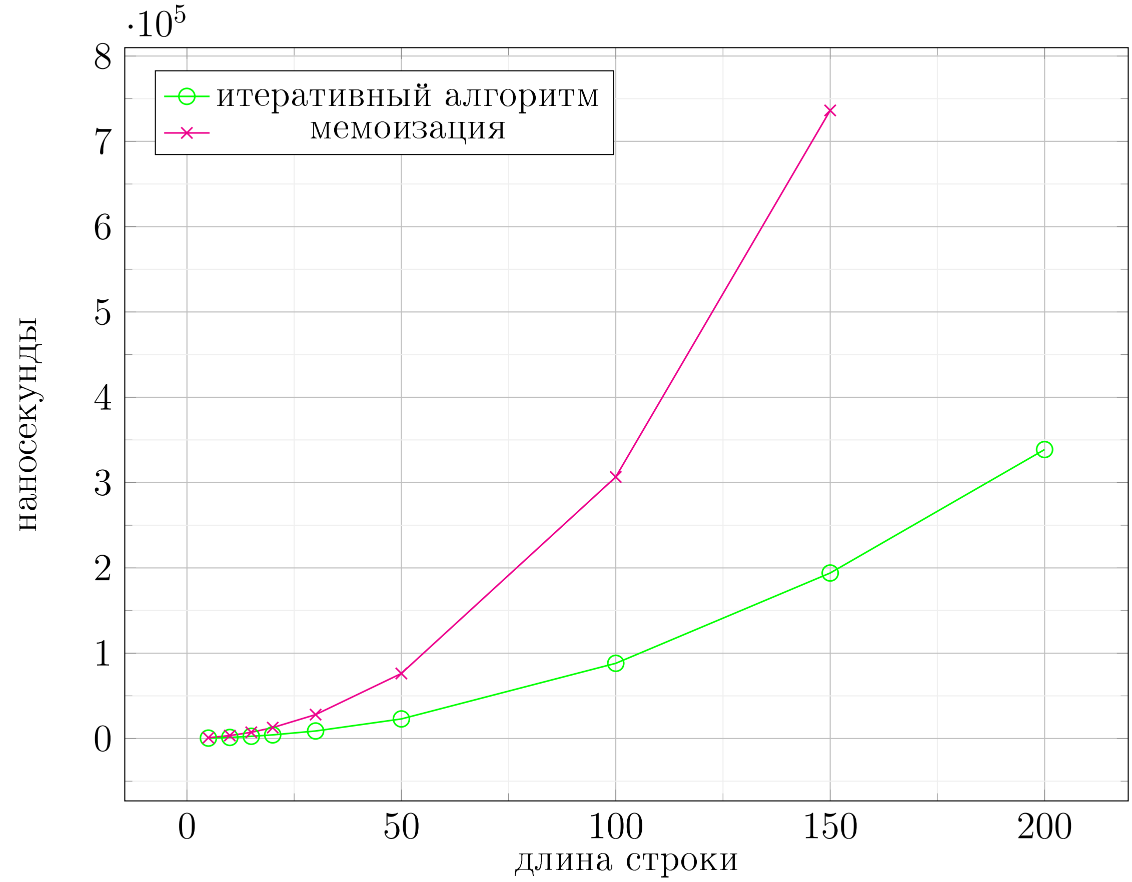

Рисунок 5.2 – Сравнение рекурсивного и итеративного матричного расстояния Левенштейна

5.3 Использование памяти

Итеративные алгоритмы Левенштейна и Дамерау – Левенштейна используют одинаковое количество памяти, следовательно, смысла их анализировать нет. Более информативным будет исследование рекурсивных и матричных реализаций этих алгоритмов. Максимальная глубина стека при вызове рекурсивных функций имеет следующий вид:

$$M_{recursive} = (n \cdot lvar + ret + ret_{int}) \cdot depth \tag{5.1}$$

Где:

n – количество аллоцированных локальных переменных;
$lvar$ – размер переменной типа int
$ret - -$адрес возврата;
$ret_{int} - -$возвращаемое значение;
$depth$ – максимальная глубина стека вызова, которая равна $|S_1| + |S_2|$.

Использование памяти при итеративных реализациях:

$$M_{iter} = |S_1| + |S_2| + (|S_1| + 1 \cdot |S_2| + 1) \cdot lvar + n \cdot lvar + ret + ret_{int} \quad (5.2)$$

Где $(|S_1| + 1 \cdot |S_2| + 1) \cdot lvar$ – место в памяти под матрицу расстояний.

5.4 Вывод

Рекурсивный алгоритм Левенштейна работает дольше итеративных реализаций – время этого алгоритма увеличивается в геометрической прогрессии с ростом размера строк. Рекурсивный алгоритм с мемоизацией превосходит простой рекурсивный алгоритм по времени. Расстояние Дамерау – Левенштейна по результатом замеров практически не отличается от расстояния Левенштейна. Однако, в системах автоматического исправления текста, где транспозиция встречается чаще, расстояние Дамерау – Левенштейна будет наиболее эффективным алгоритмом. По расходу памяти все реализации проигрывают рекурсивной за счет большого количества выделенной памяти под матрицу расстояний. Стоит отметить, что для языков, где возможна передача указателя на массивы, самым эффективным и по времени, и по памяти будет алгоритм, использующий мемоизацию.

6 Заключение

В рамках лабораторной работы были рассмотрены два алгоритма нахождения редакторского расстояния – расстояние Левенштейна и расстояние Дамерау – Левенштейна. Во время аналитического изучения алгоритмов были выявлены смысловые различия между двумя алгоритмами – расстояние Дамерау – Левенштейна более эффективно в системах автоматической замены текста, где наиболее часто встречающаяся редакторская операция – это транспозиция. В других случаях, если алгоритмы работают не с буквами в естественном языке, рациональнее использовать алгоритм расстояние Левенштейна. Самая оптимальная реализация по памяти – рекурсивный алгоритм, самая оптимальная реализация по времени – итеративный алгоритм, использующий таблицу расстояний. Для языков, где возможна передача указателя на массивы, самым эффективным по времени и по памяти будет алгоритм, использующий мемоизацию. В ходе лабораторной работы получены навыки динамического программирования, реализованы изученные алгоритмы.

7 Список использованных источников

[1] *Go 1 Release Notes*. URL: `https://golang.org/doc/go1#rune` (дата обр. 06.09.2021).

[2] *testing*. URL: `https://pkg.go.dev/testing` (дата обр. 07.09.2021).