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 ________________ Ковель А.Д.

Группа Подпись, дата И. О. Фамилия

Преподаватель ________________ Волкова Л.Л.

Подпись, дата И. О. Фамилия

Москва, 2022 г.

Оглавление

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. Пусть даны строки s1 = s1[1..L1], s2 = s2[1..L2], s1[1..i] - подстрока s1 длинной i, начиная с 1-го символа, s2[1..j] - подстрока s2 длинной j, начиная с 1-го символа. Расстояние Левентштейна посчитывается следующей формулой:

$$D(s1[1..i], s2[1..j]) = \begin{cases} 0 & i = 0, j = 0 \\ i & i > 0, j = 0 \\ j, & j > 0, i = 0 \\ min(D(s1[1..i], s2[1..j-1]) + 1 & \\ , min(D(s1[1..i-1], s2[1..j]) + 1, & \\ , min(D(s1[1..i-1], s2[1..j]) + 1 & \\ + \left[\begin{array}{ll} 0, & s1[i] = s2[j] \\ 1 & \end{array}\right. & \end{cases} \tag{2.1}$$

2.2 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:

$$f_{X,Y}(i,j) = \begin{cases} \delta_i & j = 0 \\ \delta_j & i = 0 \\ min \begin{cases} \alpha(x_i, y_i) + f_{X,Y}(i-1, j-1) \\ \delta + f_{X,Y}(i-1, j) \\ \delta + f_{X,Y}(i, j-1) \\ \begin{bmatrix} \delta + f_{X,Y}(i-2, j-2) & i,j > 1 x_i = y_{j-1} x_{i-1} = y_j \\ \infty & \text{иначе;} \end{bmatrix} \end{cases} & \text{иначе.} \end{cases} \tag{2.2}$$

2.3 Рекурсивная формула

Используя условные обозначения, описанные в разделе 2.2, рекурсивная формула для нахождения расстояния Дамерау- Левенштейна $f(i,j)$ между подстроками $x_1...x_i$ и $y_1...y_j$ имеет следующий вид:

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

$f_{X,Y}$ – 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:
  $f_{X,Y} \leq f_{X,Z} + f_{Z,Y}$

2.4 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:

$$Ci,j = \begin{cases} i & j = 0 \\ j & i = 0 \\ min \begin{cases} C_{i-1,j-1} + \alpha(x_i, y_i), \\ C_{i-1,j} + 1, \\ C_{i,j-1} + 1) \end{cases} & \text{иначе.} \end{cases} \tag{2.4}$$

При решении данной задачи используется ключевая идея динамического программирования – чтобы решить поставленную задачу, требуется разбить на 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.

2.5 Рекурсивный алгоритм расстояния Дамерау-Левенштейна с мемоизацией

При реализации рекурсивного алгоритма используется мемоизация – сохранение результатов выполнения функций для предотвращения повторных вычислений. Отличие от формулы 2.4 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.

2.6 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 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 Дамерау – Левенштейна может быть модифицирован, используя мемоизацию – достаточно инициализировать матрицу значением ∞, которое будет рассмотрено в качестве флага. На рисунке 3.3 изображена схема алгоритма, использующая этот прием.

3.3 Рекурсивные алгоритмы

На рисунке 3.2 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.

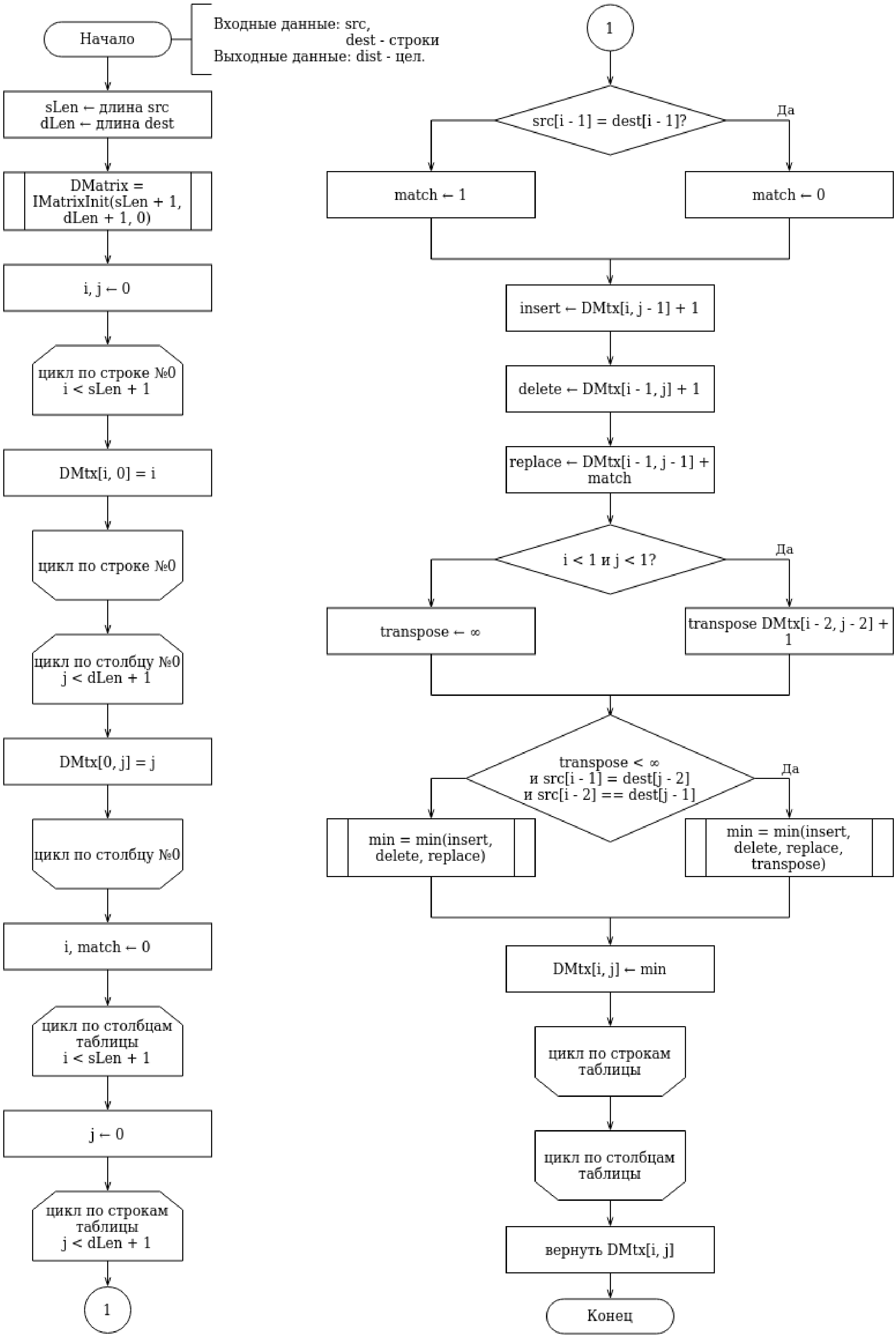

Рисунок 3.1 – Схема итерационного алгоритма расстояния Дамерау – Левенштейна с заполнением матрицы расстояний

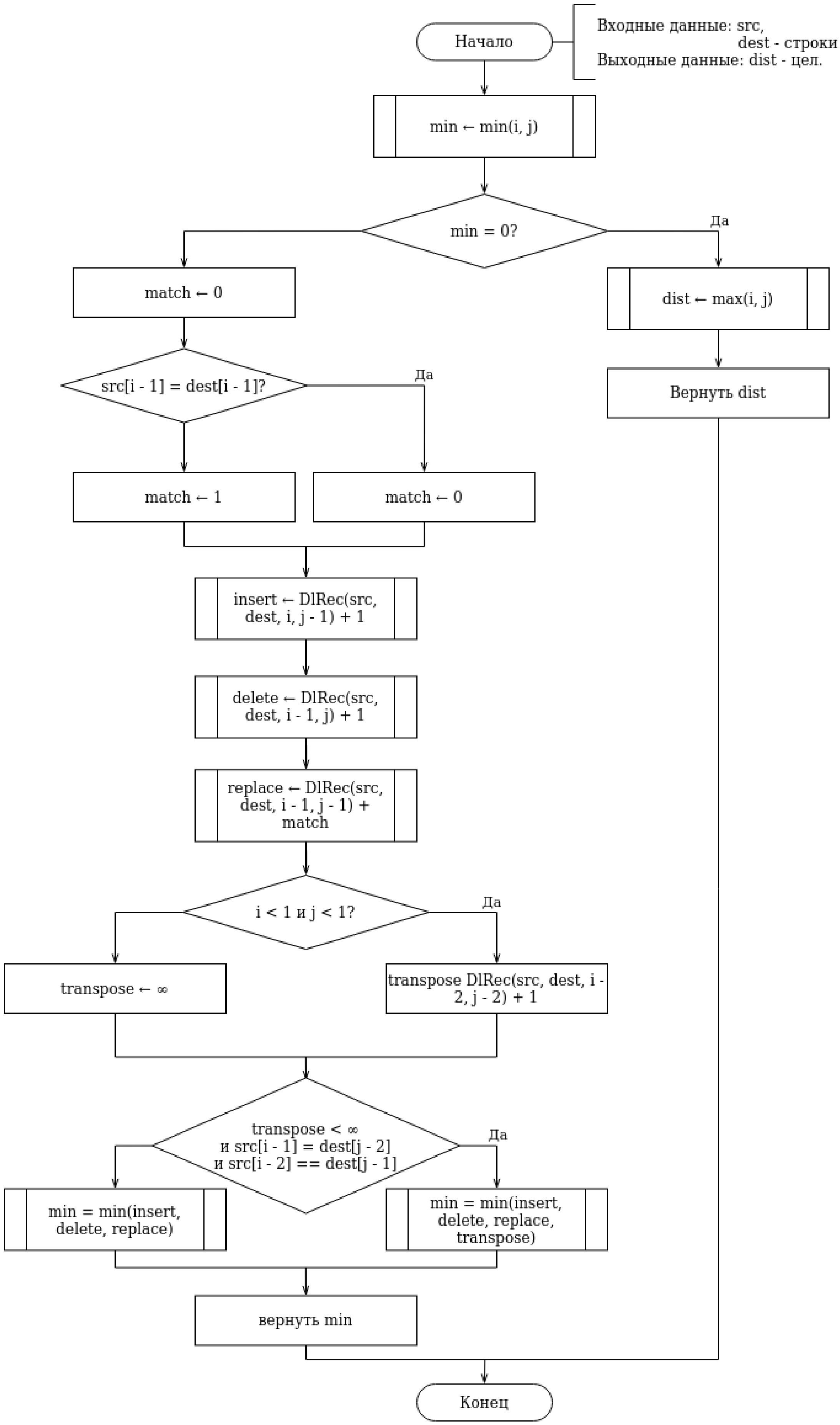

Рисунок 3.2 – Схема рекурсивного алгоритма расстояния Дамерау - Левенштейна

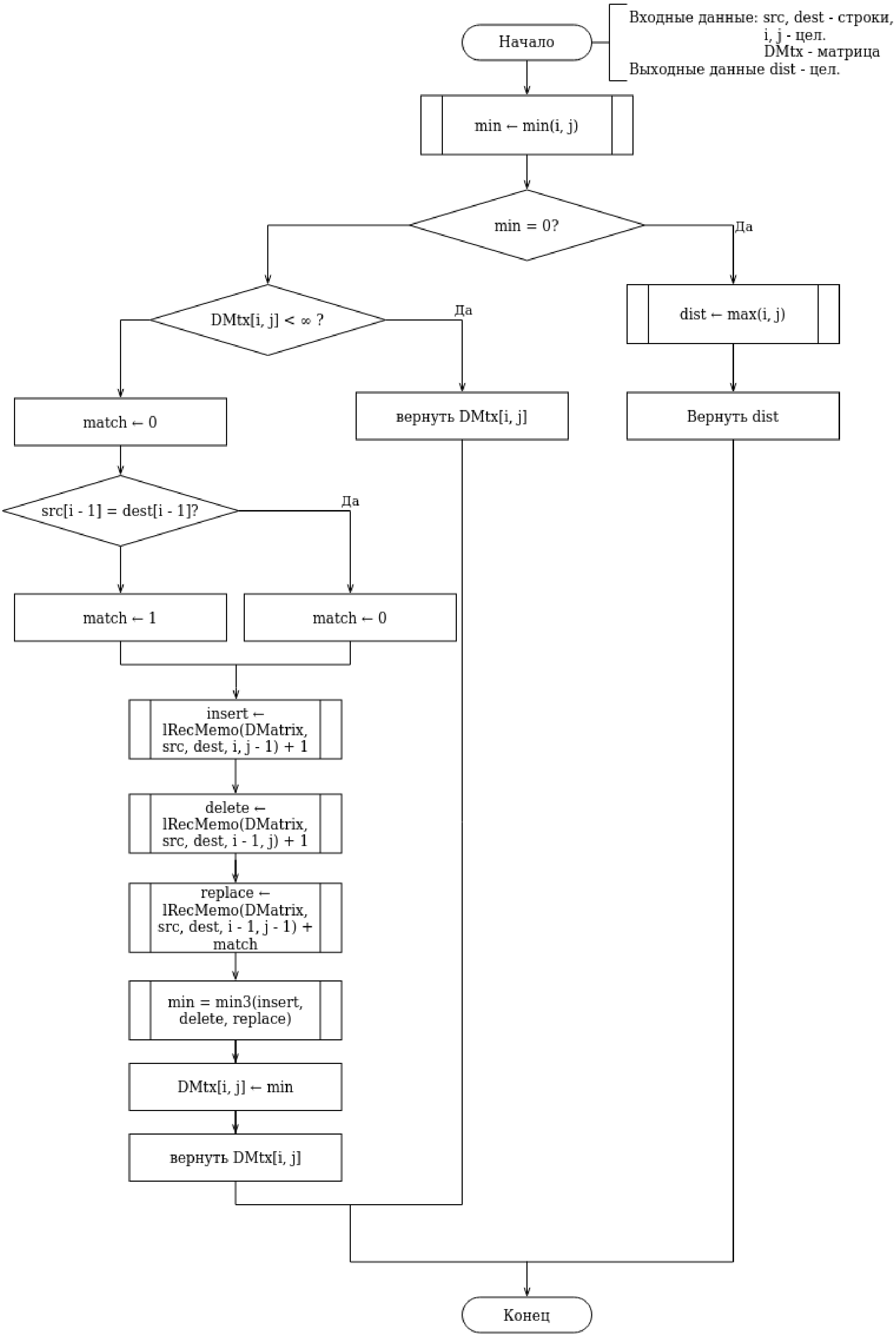

Рисунок 3.3 – Схема рекурсивного алгоритма расстояния Левенштейна с мемоизацией

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, запуск происходил через go run main.go.

4.3 Листинги кода

Реализация алгоритмов

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

В листингах **??** - 4.3 приведены реализации алгоритмов, описанных в разделе 2.

Листинг 4.1 – Программный код нахождения расстояния Дамерау – Левенштейна итеративно

```
func CountDamNoRec(src, dest string) (int, MInt) {
    var (n, m, dist, shortDist, transDist int)

    srcRune, destRune := []rune(src), []rune(dest)

    n, m = len(srcRune), len(destRune)

    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

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

            transDist = -1
            if i > 1 && j > 1 {
                transDist = distMat[i - 2][j - 2] + 1
            }

            if transDist != -1 && srcRune[i - 1] == destRune[j - 2] && srcRune[i -
                dist = getMinOfValues(insDist, delDist, eqDist, transDist)
            } else {
                dist = getMinOfValues(insDist, delDist, eqDist)
            }
            distMat[i][j] = dist
        }
    }

    shortDist = distMat[n][m]

    return shortDist, distMat
}
```

Листинг 4.2 – Программный код нахождения расстояния Дамерау – Левенштейна рекурсивно

```
func _countDamRecElem(src, dest []rune, i, j int) int {
    if (getMinOfValues(i, j) == 0) {
        return getMax0f2Values(i, j)
    }

    match := 1
```

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

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

    transpose := -1

    if i > 1 && j > 1 {
        transpose = _countDamRecElem(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 = getMinOfValues(insert, delete, replace, transpose)
    } else {
        min = getMinOfValues(insert, delete, replace)
    }
    return min
}

func CountDamRecNoCache(src, dest string) int {

    srcRune, destRune := []rune(src), []rune(dest)
    n, m := len(srcRune), len(destRune)

    return _countDamRecElem(srcRune, destRune, n, m)
}
```

Листинг 4.3 – Программный код нахождения расстояния Дамерау – Левенштейна рекурсивно с кэшем

```
func _countDamRecElemCache(src, dest []rune, i, j int, distMat MInt) int {
    if (getMinOfValues(i, j) == 0) {
        return getMaxOf2Values(i, j)
    }

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

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

```

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

    transpose := -1

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

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

    distMat[i][j] = min
    return distMat[i][j]

}

func CountDamRecCache(src, dest string) int {
    srcRune, destRune := []rune(src), []rune(dest)
    n, m := len(srcRune), len(destRune)

    distMat := makeMatrixInf(n, m)

    return _countDamRecElemCache(srcRune, destRune, n, m, distMat)
}
```

Утилиты

В листингах 4.4 - 4.6 приведены используемые утилиты.

Листинг 4.4 – Функция нахождения минимума из N целых чисел

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

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

```

```
        return min
    }
```

Листинг 4.5 – Функция нахождения максимума из двух целых чисел

```
    func getMaxOf2Values(v1, v2 int) int {
        if v1 < v2 {
            return v2
        }
        return v1
    }
```

Листинг 4.6 – Определение типа целочисленной матрицы; его инициализация и вывод

```
    type MInt [][]int

    func makeMatrix(n, m int) MInt {
        matrix := make(MInt, n + 1)

        for i := range matrix {
            matrix[i] = make([]int, m + 1)
        }

        for i := 0; i < m + 1; i++ {
            matrix[0][i] = i
        }

        for i := 0; i < n + 1; i++ {
            matrix[i][0] = i
        }
        return 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")
        }
}
```

4.4 Тестовые данные

№	S_1	S_2	DLIter	DLRec	DLRecCache
1	« »	« »	0	0	0
2	«book»	«bosk»	1	1	1
3	«book»	«back»	2	2	2
4	«book»	«bacc»	3	3	3
5	«aboba»	«acacb»	4	4	4
6	«дверь»	«деврь»	1	1	1
6	«дверь»	«дверь»	1	1	1

4.5 Вывод

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

Contents