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

Страница

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:

- delete$(u, \varepsilon) = \delta$
- insert$(\varepsilon, v) = \delta$
- 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:

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

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:

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

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:

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

$f_{X,Y}$ – 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:

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

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.

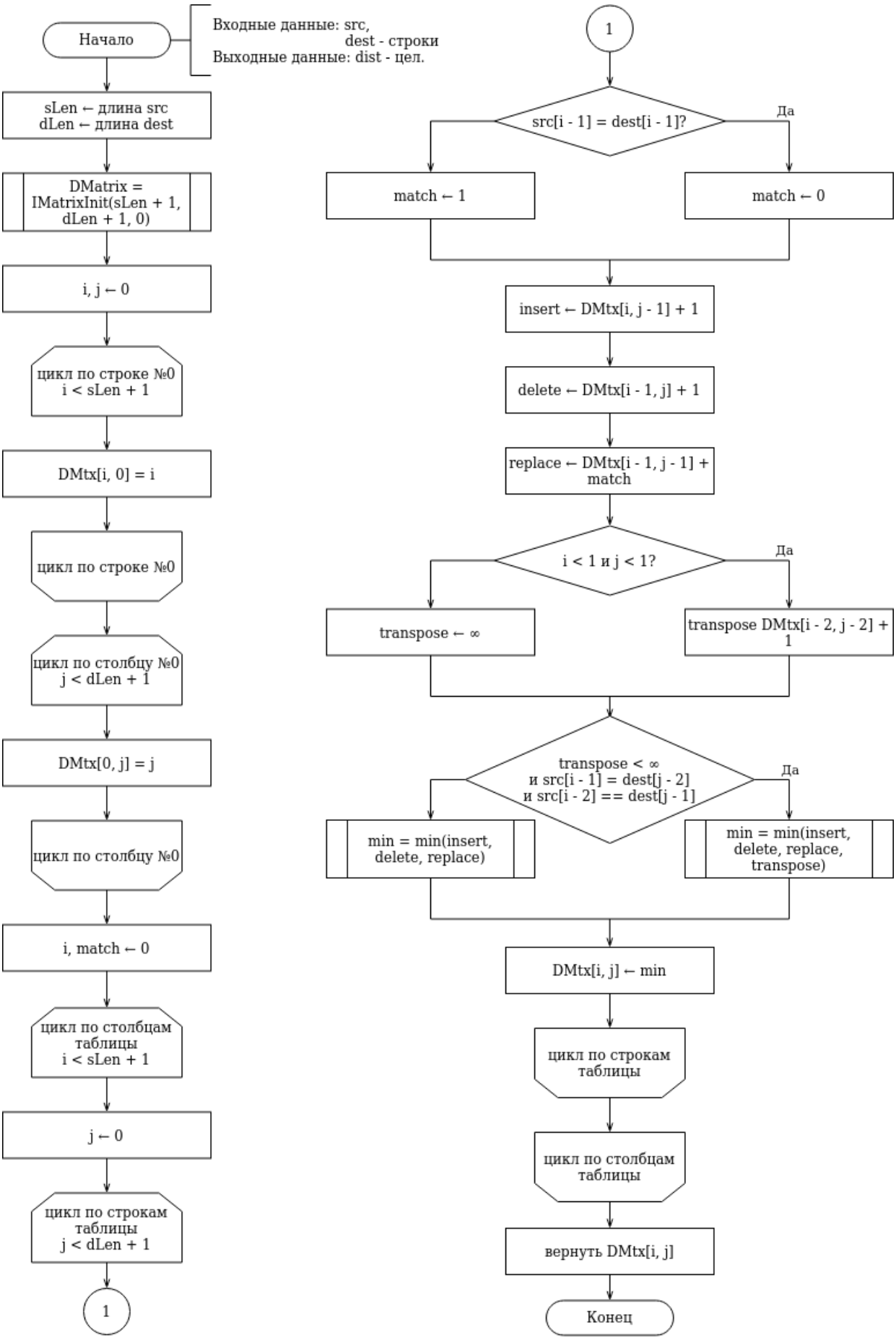

Рисунок 3.1 – Схема итерационного алгоритма расстояния Дамерау – Левенштейна с заполнением матрицы расстояний

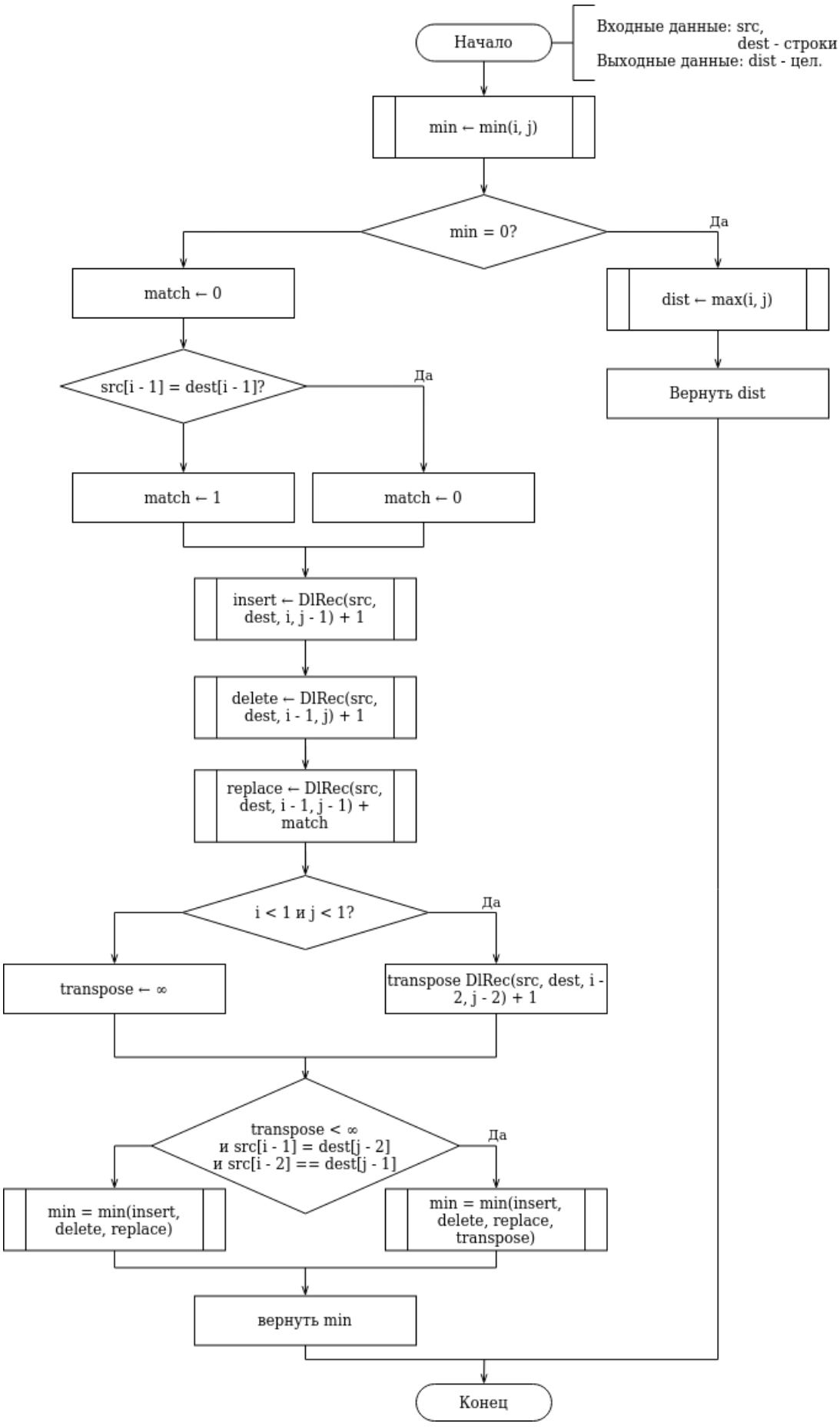

Рисунок 3.2 – Схема рекурсивного алгоритма расстояния Дамерау - Левенштейна

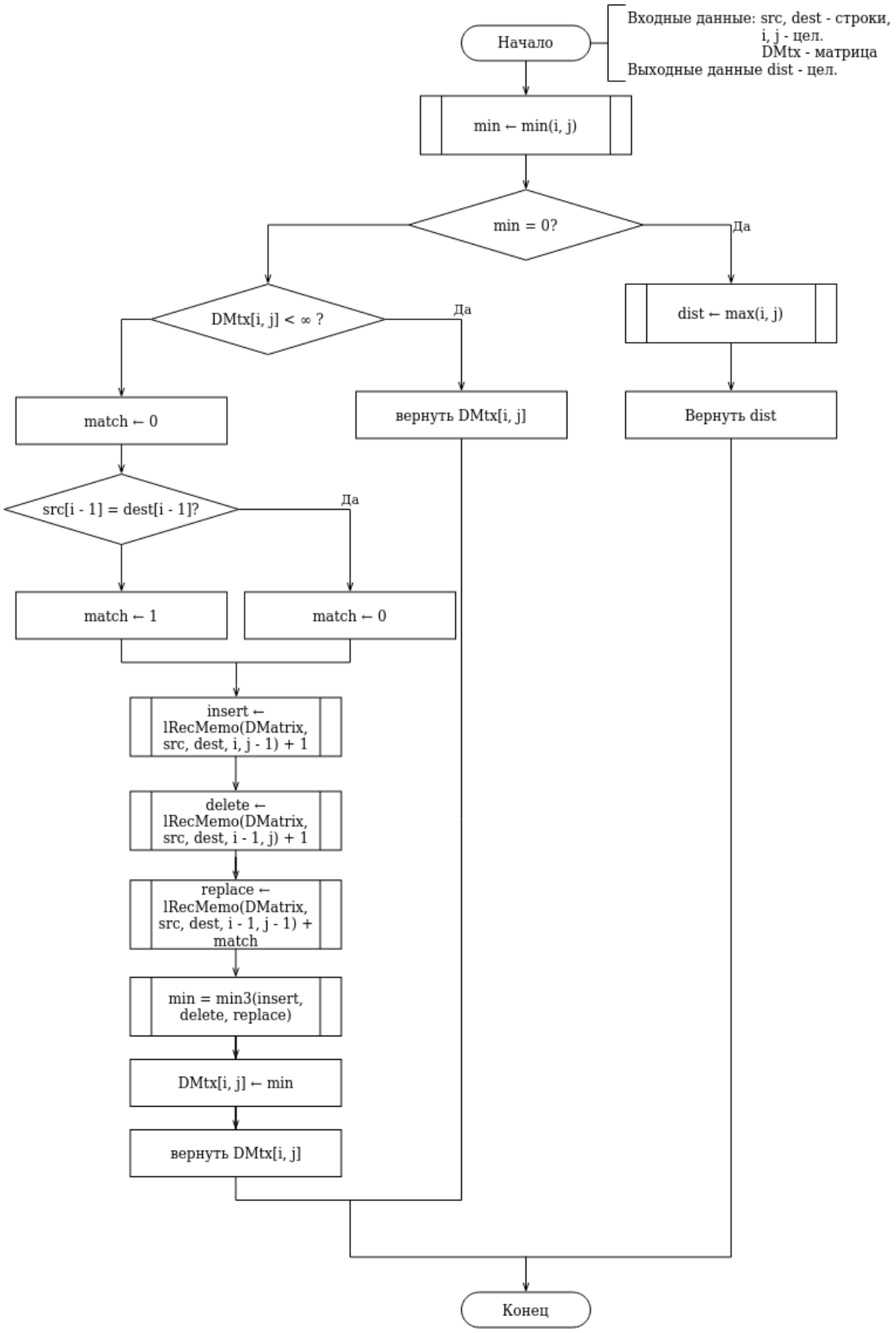

Рисунок 3.3 – Схема рекурсивного алгоритма расстояния Левенштейна с мемоизацией

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, написание сценариев осуществлялось утилитой make.

4.3 Листинги кода

Реализация алгоритмов

В качестве представления строковых данных был выбран тип rune[**rune**] – 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)

```

```
4           ans := _lRecursiveHelper(src, dest, srcLen, destLen)
5
6           return ans
7       }
8
9       func _lRecursiveHelper(src, dest []rune, i, j int) int {
10          if (utils.MinOfInt(i, j) == 0) {
11              return utils.Max2Int(i, j)
12          }
13
14          match := 1
15          if (src[i - 1] == dest[j - 1]) {
16              match = 0
17          }
18
19          insert := _lRecursiveHelper(src, dest, i, j - 1) + 1
20          delete := _lRecursiveHelper(src, dest, i - 1, j) + 1
21          replace := match + _lRecursiveHelper(src, dest, i - 1, j - 1)
22
23          min := utils.MinOfInt(insert, delete, replace)
24          return min
25      }
```

Листинг 4.4 – Программный код нахождения расстояния Дамерау – Левенштейна итеративно

```
1       func DamerauLevenshteinIterative(src, dest []rune) int{
2           srcLen, destLen := len(src), len(dest)
3
4           rows, cols := srcLen + 1, destLen + 1
5           DMtx := matrix.IMtxInit(rows, cols, 0)
6
7           for i := 1; i < rows; i++ {
8               DMtx[i][0] = i
9           }
10          for j := 1; j < cols; j++ {
11              DMtx[0][j] = j
12          }
13
14          match := 0
15          min := 0
16          for i := 1; i < rows; i++ {
17              for j := 1; j < cols; j++ {
18                  if src[i - 1] == dest[j - 1] {
19                      match = 0
20                  } else {
21                      match = 1
22                  }
```

```
23
24                  insert := DMtx[i][j - 1] + 1
25                  delete := DMtx[i - 1][j] + 1
26                  replace := DMtx[i - 1][j - 1] + match
27                  transpose := -1
28
29                  if i > 1 && j > 1 {
30                      transpose = DMtx[i - 2][j - 2] + 1
31                  }
32
33                  if i > 1 && j > 1 && src[i - 1] == dest[j - 2]
34                  && src[i - 2] == dest[j - 1] {
35                      min = utils.MinOfInt(insert, delete, replace, transpose)
36                  } else {
37                      min = utils.MinOfInt(insert, delete, replace)
38                  }
39                  DMtx[i][j] = min
40              }
41          }
42          DMtx[srcLen][destLen] = min
43          return DMtx[srcLen][destLen]
44      }
```

Листинг 4.5 – Программный код нахождения расстояния Дамерау – Левенштейна рекурсивно

```
1       func DamerauLevenshteinRecursive(src, dest []rune) int {
2           srcLen, destLen := len(src), len(dest)
3
4           ans := _dRecursiveHelper(src, dest, srcLen, destLen)
5
6           return ans
7       }
8
9       func _dRecursiveHelper(src, dest []rune, i, j int) int {
10          if (utils.MinOfInt(i, j) == 0) {
11              return utils.Max2Int(i, j)
12          }
13
14          match := 1
15          if (src[i - 1] == dest[j - 1]) {
16              match = 0
17          }
18
19          insert := _dRecursiveHelper(src, dest, i, j - 1) + 1
20          delete := _dRecursiveHelper(src, dest, i - 1, j) + 1
21          replace := match + _dRecursiveHelper(src, dest, i - 1, j - 1)
22
```

```
        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
```

```
2
3     func IMtxInit(rows, cols, filler int) IMtx {
4         mtx := make(IMtx, rows)
5         for i := range mtx {
6             mtx[i] = make([]int, cols)
7             for j := 0; j < cols; j++ {
8                 mtx[i][j] = filler
9             }
10        }
11        return mtx
12    }
13
14    func IMtxLog(mtx IMtx, rows, cols int) {
15        for i := 0; i < rows; i++ {
16            for j := 0; j < cols; j++ {
17                fmt.Print(mtx[i][j])
18                fmt.Print(" ")
19            }
20            fmt.Println("")
21        }
22    }
```

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.

Contents