



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: _____ Расстояния Левенштейна и Дamerau – Левенштейна

Дисциплина: _____ Анализ алгоритмов

Студент	<u>ИУ7-56Б</u>	_____	<u>Ковель А.Д.</u>
	Группа	Подпись, дата	И. О. Фамилия
Преподаватель		_____	<u>Волкова Л.Л.</u>
Преподаватель		_____	<u>Строганов Ю.В.</u>
		Подпись, дата	И. О. Фамилия

Москва, 2022 г.

Оглавление

Введение	4
1 Аналитический раздел	6
1.1 Расстояние Левенштейна	6
1.2 Расстояние Дамерау – Левенштейна	7
1.3 Рекурсивная формула	7
1.4 Матрица расстояний	9
1.5 Рекурсивный алгоритм расстояния Дамерау – Левенштейна с мемоизацией	9
1.6 Вывод	10
2 Конструкторский раздел	11
2.1 Матричные итерационные алгоритмы	11
2.2 Модификация матричных алгоритмов	11
2.3 Разработка алгоритмов	11
2.4 Вывод	17
3 Технологический раздел	18
3.1 Требования к ПО	18
3.2 Средства реализации	18
3.3 Листинги кода	18
3.3.1 Реализация алгоритмов	18
3.3.2 Подпрограммы	22
3.4 Тестовые данные	23
3.5 Вывод	23
4 Исследовательская часть	24
4.1 Технические характеристики	24
4.2 Демонстрация работы программы	24
4.3 Время выполнения алгоритмов	25
4.4 Использование памяти	27

4.4.1	Нерекурсивный алгоритм поиска расстояния Дамерау-Левенштейна	27
4.4.2	Рекурсивный алгоритм поиска расстояния без кэша Дамерау-Левенштейна	28
4.4.3	Рекурсивный алгоритм поиска расстояния с использованием кэша Дамерау-Левенштейна	29
4.5	Вывод	30
Заключение		31
Список использованных источников		32

Введение

Цель лабораторной работы — разработка, реализация и исследование алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна.

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 [1], впоследствии её связали с его именем. В данной работе будут рассмотрены алгоритмы редакционного расстояния Левенштейна и расстояние Дамерау — Левенштейна [2].

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

Задачи данной лабораторной следующие:

- 1) изучение расстояний Левенштейна и Дamerau-Левенштейна;
- 2) применение метода динамического программирования для реализации алгоритма;
- 3) получение практических навыков реализаций алгоритма Левенштейна и Дamerau-Левенштейна;
- 4) проведение сравнительного анализа алгоритмов определения расстояния между строками по затратам времени и памяти;
- 5) получение экспериментального подтверждения различий по временной эффективности алгоритмов расстояния между строками, путем измерения процессорного времени с помощью разработанного программного обеспечения;
- 6) описание и обоснование полученных результатов в отчете о выполненной лабораторной работе, выполненного как расчетно-пояснительная записка к работе.

1 Аналитический раздел

В этом разделе будут представлены описания алгоритмов нахождения редакторских расстояний Левенштейна и Дамерау-Левенштейна и их практическое применение.

1.1 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, $X = x_1x_2 \dots x_n$, и, $Y = y_1y_2 \dots y_n$, расстояние между ними можно вычислить с помощью операций:

- $\text{delete}(u, \varepsilon) = \delta$;
- $\text{insert}(\varepsilon, v) = \delta$;
- $\text{replace}(u, v) = \alpha(u, v) \leq 0$ (здесь, $\alpha(u, u) = 0 \forall u$);

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 $s1$ длиной i , начиная с 1-го символа, $s2[1..j]$ — подстрока $s2$ длиной j , начиная с 1-го символа. Расстояние Левенштейна посчитывается формулой 1.1:

$$D(s1[1..i], s2[1..j]) = \begin{cases} 0, i = 0, j = 0, \\ i, i > 0, j = 0, \\ j, j > 0, i = 0, \\ \min(D(s1[1..i], s2[1..j-1]) + 1, \\ \min(D(s1[1..i-1], s2[1..j]) + 1, \\ \min(D(s1[1..i-1], s2[1..j]) + 1 \\ + \begin{cases} 0, & s1[i] = s2[j] \\ 1 \end{cases} \end{cases} \quad (1.1)$$

1.2 Расстояние Дамерау – Левенштейна

Расстояние Дамерау – Левенштейна — модификация расстояния Левенштейна, добавляющая транспозицию к редакторским операциям, предложенными Левенштейном (см. 1.1). изначально алгоритм разрабатывался для сравнения текстов, набранных человеком (Дамерау показал, 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 1.1, рекурсивная формула для нахождения расстояния Дамерау – Левенштейна, $f(i, j)$, между подстроками, $x_1 \dots x_i$ и, $y_1 \dots y_j$, имеет следующий вид 1.3:

$$f_{X,Y}(i, j) = \begin{cases} \delta_i, & j = 0, \\ \delta_j, & i = 0, \\ \min \begin{cases} \alpha(x_i, y_i) + f_{X,Y}(i-1, j-1) \\ \delta + f_{X,Y}(i-1, j) \\ \delta + f_{X,Y}(i, j-1) \\ \delta + f_{X,Y}(i-2, j-2), & \text{если } i, j > 1; \\ & x_i = y_{j-1}; \\ & x_{i-1} = y_j \\ \infty, & \text{иначе} \end{cases} \end{cases} \quad (1.2)$$

1.3 Рекурсивная формула

Используя условные обозначения, описанные в разделе 1.2, рекурсивная формула для нахождения расстояния Дамерау – Левенштейна $f(i, j)$ между

подстроками, $x_1 \dots x_i$, и, $y_1 \dots y_j$, имеет вид 1.3:

$$f_{X,Y}(i, j) = \begin{cases} \delta_i, & j = 0, \\ \delta_j, & i = 0, \\ \min \begin{cases} \alpha(x_i, y_i) + f_{X,Y}(i-1, j-1) \\ \delta + f_{X,Y}(i-1, j) \\ \delta + f_{X,Y}(i, j-1) \\ \delta + f_{X,Y}(i-2, j-2), & \text{если } i, j > 1; \\ & x_i = y_{j-1}; \\ & x_{i-1} = y_j \\ \infty, & \text{иначе} \end{cases} \end{cases} \quad (1.3)$$

$f_{X,Y}$ — редакционное расстояние между двумя подстроками — первыми i символами строки X и первыми j символами строки Y . Можно вывести 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:

$$f_{X,Y} \geq \text{abs}(|X| - |Y|);$$

- аналогично свойству треугольника, редакционное расстояние между двумя строками не может быть больше чем редакционные расстояния каждой из этих строк с третьей:

$$f_{X,Y} \leq f_{X,Z} + f_{Z,Y}.$$

1.4 Матрица расстояний

В алгоритме нахождения редакторского расстояния Дамерау – Левенштейна возможно использование матрицы расстояний.

Пусть, $C_{0...|X|,0...|Y|}$, — матрица расстояний, где, $C_{i,j}$ — минимальное количество редакторских операций, необходимое для преобразования подстроки, $x_1 \dots x_i$, в подстроку, $y_1 \dots y_j$. Матрица заполняется следующим образом 1.4:

$$C_{i,j} = \begin{cases} i & j = 0, \\ j & i = 0, \\ \min \begin{cases} C_{i-1,j-1} + \alpha(x_i, y_i), \\ C_{i-1,j} + 1, \\ C_{i,j-1} + 1 \end{cases} & \end{cases} \quad (1.4)$$

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.

1.5 Рекурсивный алгоритм расстояния Дамерау – Левенштейна с мемоизацией

При реализации рекурсивного алгоритма используется мемоизация — сохранение результатов выполнения функций для предотвращения повторных вычислений. Отличие от формулы 1.4 состоит лишь в начальной инициализации матрицы флагом ∞ , который сигнализирует о том, была ли обработана ячейка. В случае если ячейка была обработана, алгоритм переходит к следующему шагу.

1.6 Вывод

Были рассмотрены обе вариации алгоритма редакторского расстояния (Левенштейна и Дамерау–Левенштейна). Также были приведены разные способы реализации этих алгоритмов такие как: рекурсивный, итеративный и рекурсивный с мемоизацией. Итеративный может быть реализован с помощью парадигм динамического программирования или матрицей расстояния. Рекурсивный алгоритм с мемоизацией позволяет ускорить обычный рекурсивный алгоритм за счет матрицы, в которой промежуточные подсчеты.

2 Конструкторский раздел

В данном разделе представлены схемы реализуемых алгоритмов и их модификации.

2.1 Матричные итерационные алгоритмы

На рисунке 2.2 изображена схема алгоритма нахождения расстояния Дамерау – Левенштейна итеративно с использованием матрицы расстояний.

2.2 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 2.4 изображена схема алгоритма, использующая этот прием.

2.3 Разработка алгоритмов

На рисунке 2.5 изображена схема рекурсивного алгоритма нахождения расстояния Дамерау – Левенштейна.

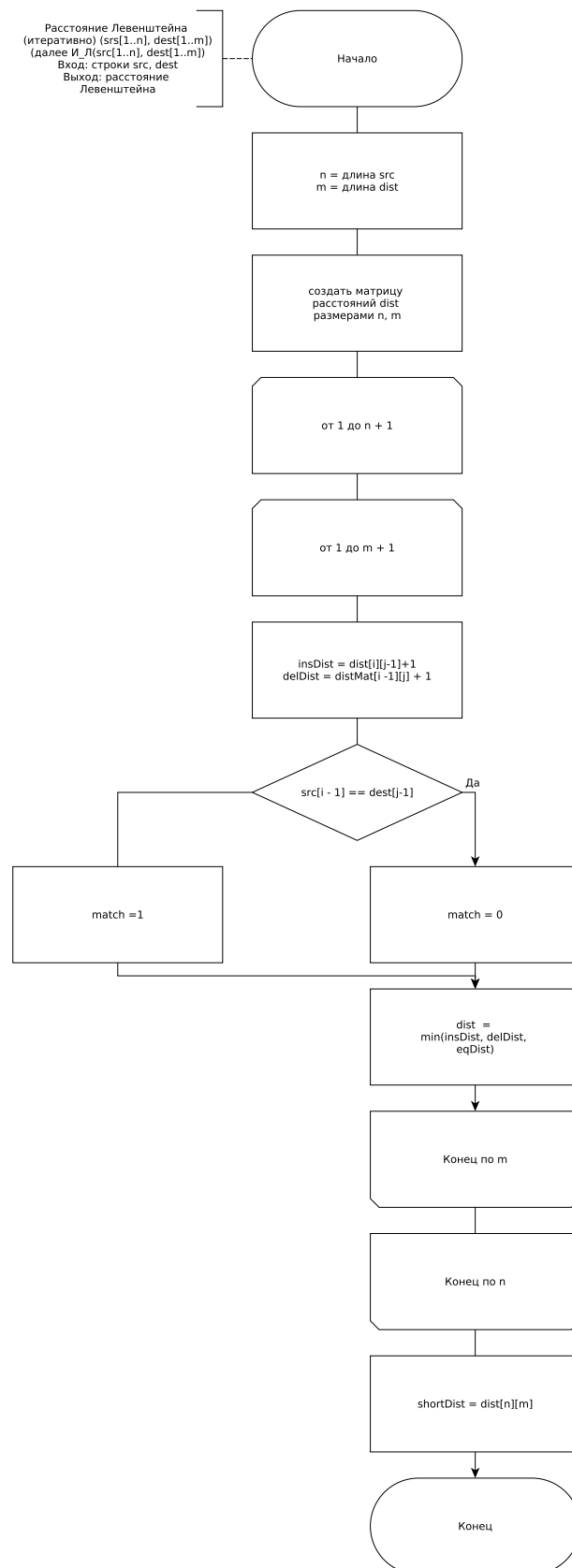


Рисунок 2.1 – Схема итерационного алгоритма расстояния Левенштейна с заполнением матрицы расстояний

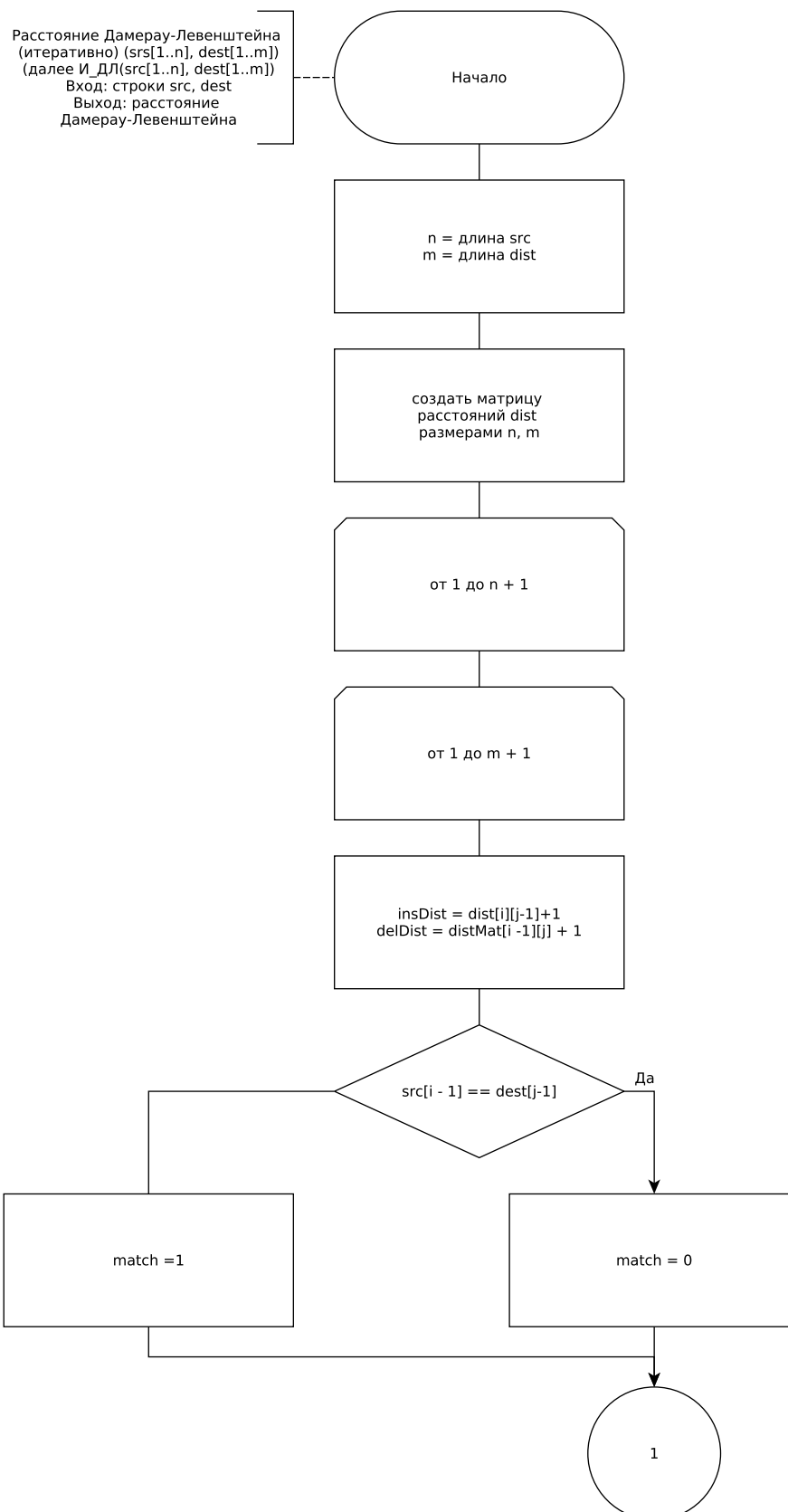


Рисунок 2.2 – Схема итерационного алгоритма расстояния Дамерау – Левенштейна с заполнением матрицы расстояний

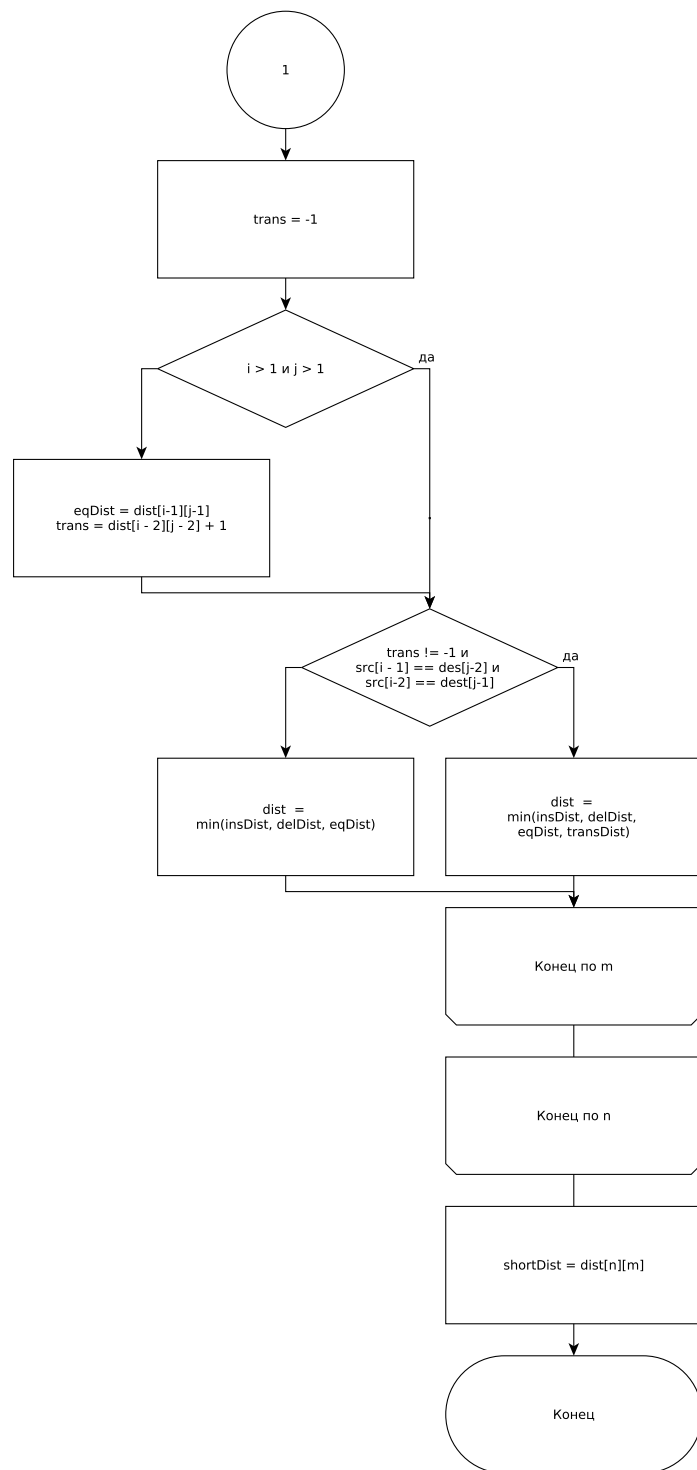


Рисунок 2.3 – Схема итерационного алгоритма расстояния Дамерау – Левенштейна с заполнением матрицы расстояний

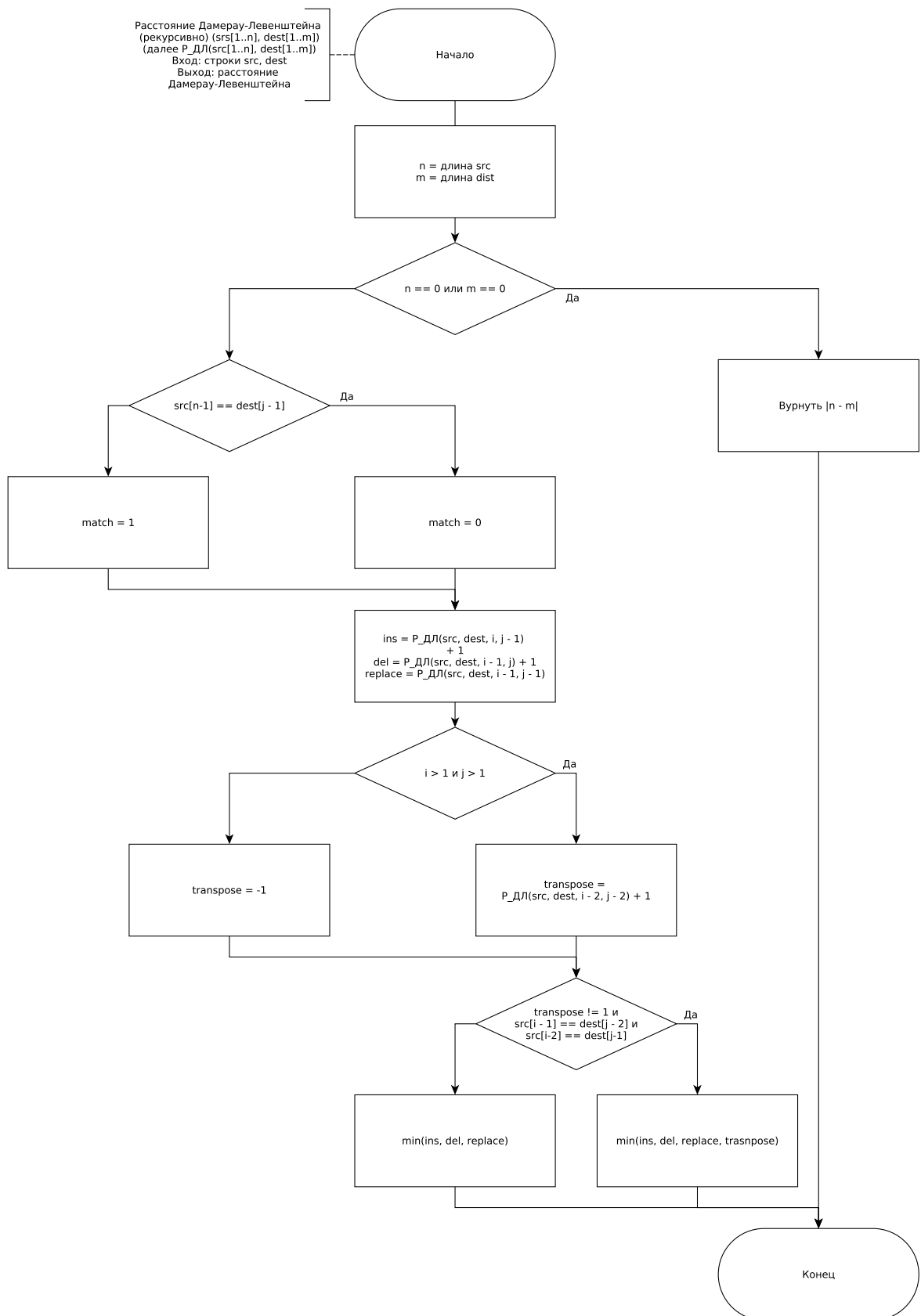


Рисунок 2.4 – Схема рекурсивного алгоритма расстояния Дамерау–Левенштейна с мемоизацией

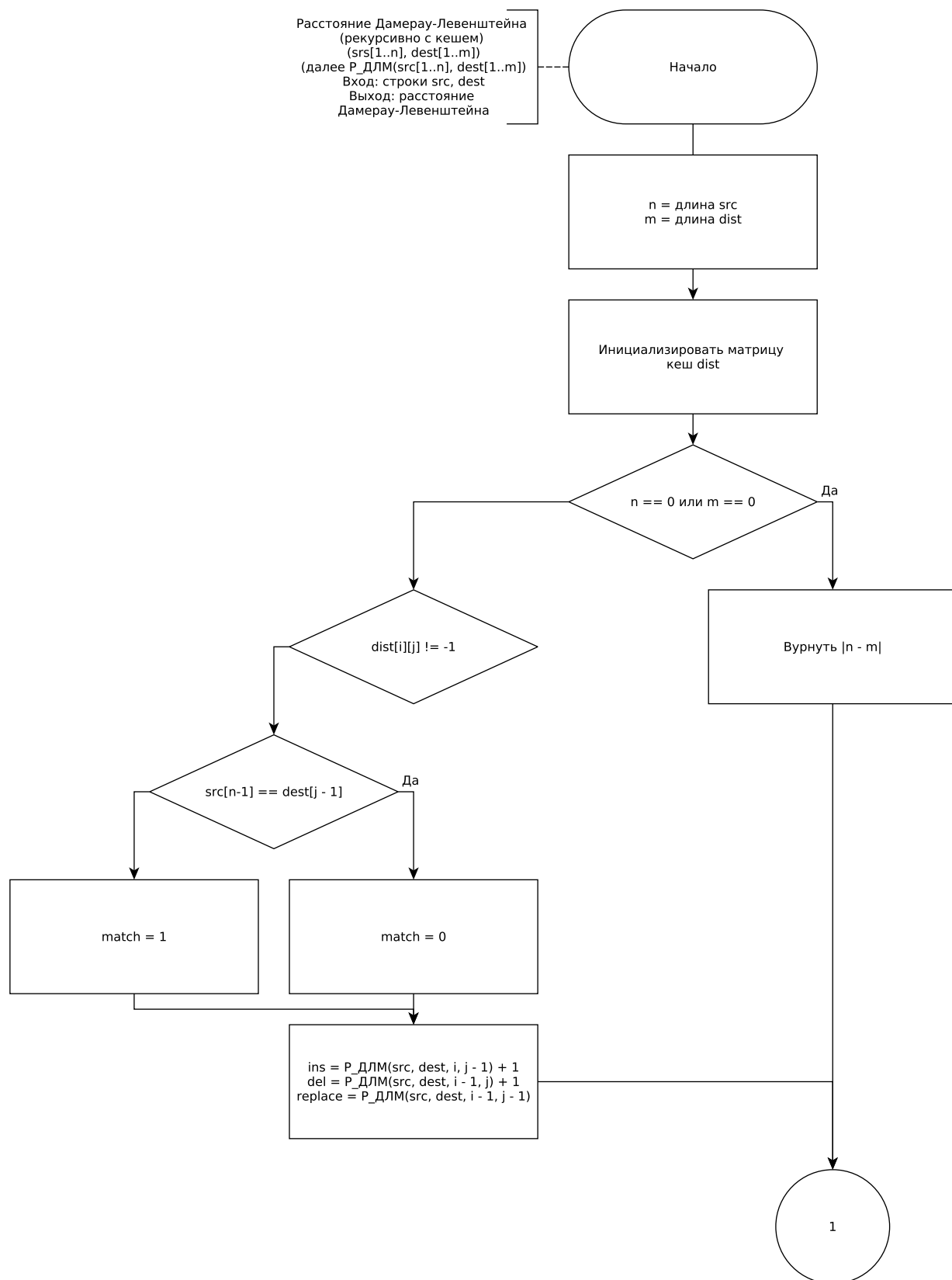


Рисунок 2.5 – Схема рекурсивного алгоритма расстояния
Дамерау – Левенштейна

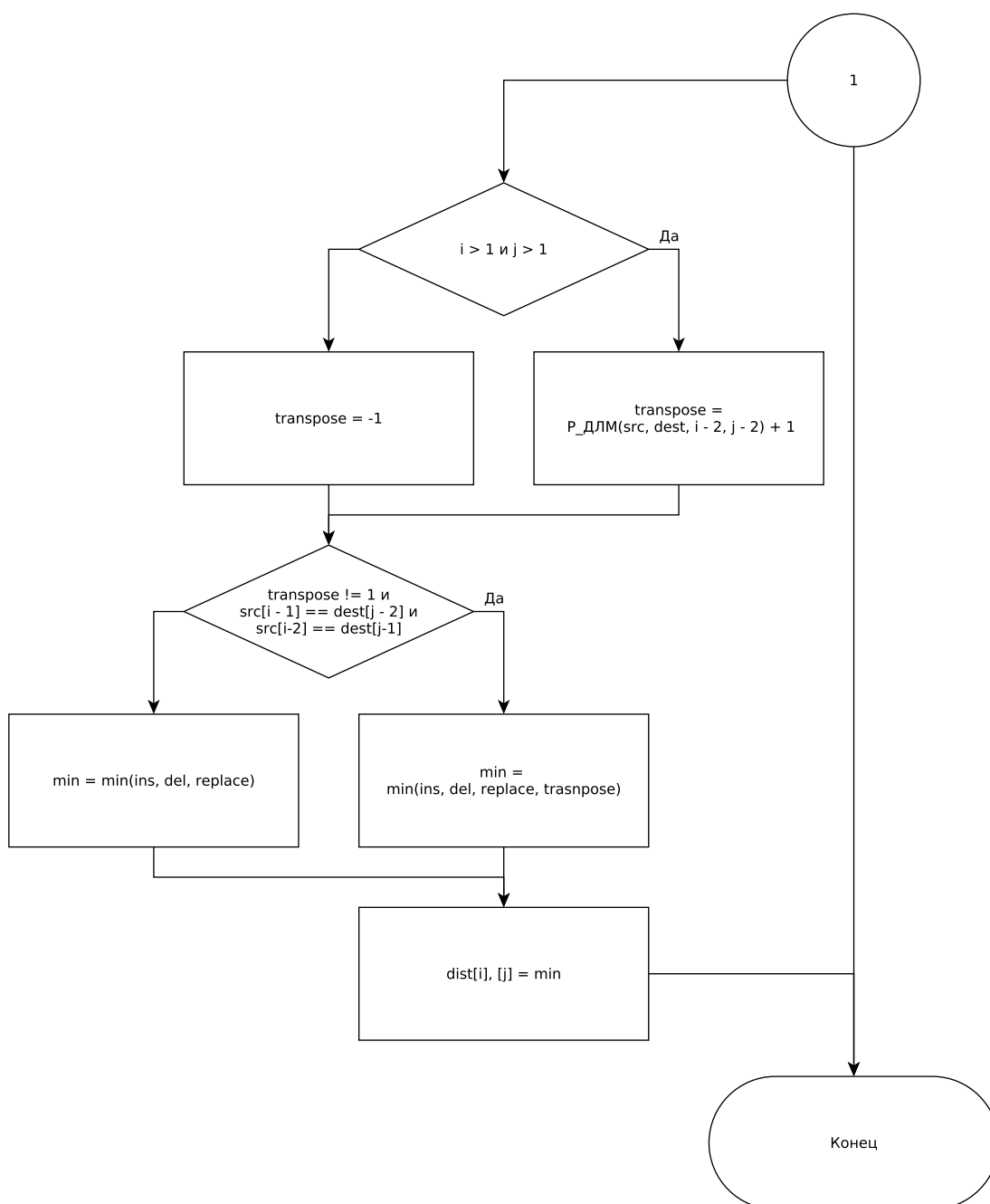


Рисунок 2.6 – Схема рекурсивного алгоритма расстояния Дамерау – Левенштейна

2.4 Вывод

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

3 Технологический раздел

В данном разделе будут приведены требования к программному обеспечению, средства реализации и листинга кода.

3.1 Требования к ПО

Программа должна отвечать следующим требованиям:

- в программе возможно измерение процессорного времени;
- программа принимает на вход две строки;
- программа выдает редакторские расстояние Левенштейна и Дамерау – Левенштейна.

3.2 Средства реализации

Для реализации ПО был выбран язык программирования Golang[3].

В данном языке есть все требующиеся инструменты для данной лабораторной работы.

В качестве среды разработки была выбрана среда VS Code[4], запуск происходил через команду `go run main.go`.

3.3 Листинги кода

В листингах представлены реализации алгоритмов поиска редакторских расстояний.

3.3.1 Реализация алгоритмов

В листингах 3.1 - 3.3 приведены реализации алгоритмов, описанных в разделе 1.

Листинг 3.1 – Программный код нахождения расстояния
Дамерау – Левенштейна итеративно

```
1 func CountDamNoRec(src, dest string) (int, MInt) {
2
3     var (n, m, dist, shortDist, transDist int)
4
5     srcRune, destRune := []rune(src), []rune(dest)
6
7     n, m = len(srcRune), len(destRune)
8     distMat := makeMatrix(n, m)
9
10    for i := 1; i < n + 1; i++ {
11
12        for j := 1; j < m + 1; j++ {
13
14            insDist := distMat[i][j - 1] + 1
15            delDist := distMat[i - 1][j] + 1
16            match := 1
17
18            if src[i - 1] == dest[j - 1] {
19                match = 0
20            }
21            eqDist := distMat[i - 1][j - 1] + match
22            transDist = -1
23            if i > 1 && j > 1 {
24                transDist = distMat[i - 2][j - 2] + 1
25            }
26
27            if transDist != -1 && srcRune[i - 1] == destRune[j - 2] &&
28
29                srcRune[i - 2] == destRune[j - 1] {
30                dist = getMinOfValues(insDist, delDist, eqDist, transDist)
31            } else {
32                dist = getMinOfValues(insDist, delDist, eqDist)
33            }
34            distMat[i][j] = dist
35        }
36    }
37    shortDist = distMat[n][m]
38    return shortDist, distMat
39 }
```

Листинг 3.2 – Программный код нахождения расстояния
Дамерау – Левенштейна рекурсивно

```
1 func _countDamRecElem(src, dest []rune, i, j int) int {
2     if (getMinOfValues(i, j) == 0) {
3         return getMaxOf2Values(i, j)
4     }
5
6     match := 1
7     if (src[i-1] == dest[j-1]) {
8         match = 0
9     }
10
11     insert := _countDamRecElem(src, dest, i, j-1) + 1
12     delete := _countDamRecElem(src, dest, i-1, j) + 1
13     replace := match+_countDamRecElem(src, dest, i-1, j-1)
14
15     transpose := -1
16
17     if i > 1 && j > 1 {
18         transpose = _countDamRecElem(src, dest, i-2, j-2) + 1
19     }
20
21     min := 0
22     if transpose != -1 && src[i-1] == dest[j-2]
23     && src[i-2] == dest[j-1] {
24         min = getMinOfValues(insert, delete, replace, transpose)
25     } else {
26         min = getMinOfValues(insert, delete, replace)
27     }
28     return min
29 }
30
31 func CountDamRecNoCache(src, dest string) int {
32
33     srcRune, destRune := []rune(src), []rune(dest)
34     n, m := len(srcRune), len(destRune)
35
36     return _countDamRecElem(srcRune, destRune, n, m)
37 }
```

Листинг 3.3 – Программный код нахождения расстояния
Дамерау – Левенштейна рекурсивно с кэшем

```
1 func _countDamRecElemCache(src, dest []rune, i, j int, distMat
  MInt) int {
2   if (getMinOfValues(i, j) == 0) {
3     return getMaxOf2Values(i, j)
4   }
5   if distMat[i][j] != -1 {
6     return distMat[i][j]
7   }
8   match := 1
9   if (src[i-1] == dest[j-1]) {
10    match = 0
11  }
12  insert := _countDamRecElemCache(src, dest, i, j-1, distMat) +
    1
13  delete := _countDamRecElemCache(src, dest, i-1, j, distMat) +
    1
14  replace := match + _countDamRecElemCache(src, dest, i-1, j-
    1, distMat)
15  transpose := -1
16  if i > 1 && j > 1 {
17    transpose = _countDamRecElemCache(src, dest, i-2, j-2,
    distMat) + 1
18  }
19  min := 0
20  if transpose != -1 && src[i-1] == dest[j-2]
21  && src[i-2] == dest[j-1] {
22    min = getMinOfValues(insert, delete, replace, transpose)
23  } else {
24    min = getMinOfValues(insert, delete, replace)
25  }
26  distMat[i][j] = min
27  return distMat[i][j]
28 }
29 func CountDamRecCache(src, dest string) int {
30   srcRune, destRune := []rune(src), []rune(dest)
31   n, m := len(srcRune), len(destRune)
32   distMat := makeMatrixInf(n, m)
33   return _countDamRecElemCache(srcRune, destRune, n, m, distMat)
34 }
```

3.3.2 Подпрограммы

В листингах 3.4 - 3.6 приведены используемые подпрограммы.

Листинг 3.4 – Функция нахождения минимума из N целых чисел

```
1 func getMinOfValues(values ...int) int {
2     min := values[0]
3     for _, i := range values {
4         if min > i {
5             min = i
6         }
7     }
8     return min
9 }
```

Листинг 3.5 – Функция нахождения максимума из двух целых чисел

```
1 func getMaxOf2Values(v1, v2 int) int {
2     if v1 < v2 {
3         return v2
4     }
5     return v1
6 }
```

Листинг 3.6 – Определение типа целочисленной матрицы; его
инициализация

```
1 type MInt [][]int
2 func makeMatrix(n, m int) MInt {
3     matrix := make(MInt, n + 1)
4     for i := range matrix {
5         matrix[i] = make([]int, m + 1)
6     }
7     for i := 0; i < m + 1; i++ {
8         matrix[0][i] = i
9     }
10    for i := 0; i < n + 1; i++ {
11        matrix[i][0] = i
12    }
13    return matrix
14 }
15 }
```

Листинг 3.7 – Вывод матрицы расстояний

```

1
2 func (mat MInt) PrintMatrix() {
3     for i := 0; i < len(mat); i++ {
4         for j := 0; j < len(mat[0]); j++ {
5             fmt.Printf("%3d ", mat[i][j])
6         }
7         fmt.Printf("\n")
8     }
9 }

```

3.4 Тестовые данные

Таблица 3.1 – Тестирование строк

№	S_1	S_2	ДЛ (итер.)	ДЛ (рек.)	ДЛ (кэш)	Лев (итер.)
1	« »	« »	0	0	0	0
2	«book»	«bosk»	1	1	1	1
3	«book»	«back»	2	2	2	2
4	«book»	«bacc»	3	3	3	3
5	«aboba»	«acacb»	4	4	4	4
6	«дверь»	«девр»	1	1	1	2
6	«дверь»	«дверь»	0	0	0	0

3.5 Вывод

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

4 Исследовательская часть

В данном разделе будут приведены примеры работы программ, постановка эксперимента и сравнительный анализ алгоритмов на основе полученных данных.

4.1 Технические характеристики

Тестирование выполнялось на устройстве со следующими техническими характеристиками:

- Операционная система Pop!_OS 22.04 LTS [5] Linux [6];
- Оперативная память 16 Гб;
- Процессор AMD® Ryzen 7 2700 eight-core processor × 16 [7].

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

4.2 Демонстрация работы программы

Программа получает на вход 2 слова и выдает 4 расстояния, соответствующих алгоритмов поиска редакторского расстояния Левенштейна, итеративного Дамерау-Левенштейна, рекурсивного Дамерау-Левенштейна, рекурсивного с мемоизацией Дамерау-Левенштейна.

На рисунке 4.1 представлен результат работы программы.


```
→ stud_52 git:(lab-01) X go run main.go
Расстояние Левенштейна

Введите первое слово: first
Введите второе слово: second

Левейнштейн не рекурсивный: 6
  0  1  2  3  4  5  6
  1  1  2  3  4  5  6
  2  2  2  3  4  5  6
  3  3  3  3  4  5  6
  4  3  4  4  4  5  6
  5  4  4  5  5  5  6

Дамерау-Левенштейн не рекурсивный: 6
  0  1  2  3  4  5  6
  1  1  2  3  4  5  6
  2  2  2  3  4  5  6
  3  3  3  3  4  5  6
  4  3  4  4  4  5  6
  5  4  4  5  5  5  6

Дамерау-Левенштейн рекурсивный без кэша: 6
Дамерау-Левенштейн рекурсивный с кэшем: 6
```

Рисунок 4.1 – Пример работы программы

4.3 Время выполнения алгоритмов

Алгоритмы тестировались при помощи профилирования — сбора характеристик работы программы: времени выполнения. Для каждой функции были написаны тесты оценки эффективности, представленные библиотеками Си.

Листинг 4.1 – Пример теста эффективности

```

1 import "C"
2
3 func test(){
4
5     cputime1 := C.getThreadCpuTimeNs()
6
7     doWork()
8
9     cputime2 := C.getThreadCpuTimeNs()
10    fmt.Printf(cputime2 - cputime1)
11
12 }

```

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

Таблица 4.1 – Время выполнения алгоритмов

Длина строк	Время выполнения			
	ДЛ (кэш)	ДЛ (итер.)	Лев (итер.)	ДЛ (рек.)
5	2344	1114	1091	17228
10	6747	3142	2823	109170295
40	92218	36281	33362	-
80	402839	142910	122148	-
160	1582974	646498	499268	-
240	3505394	1348110	1122182	-

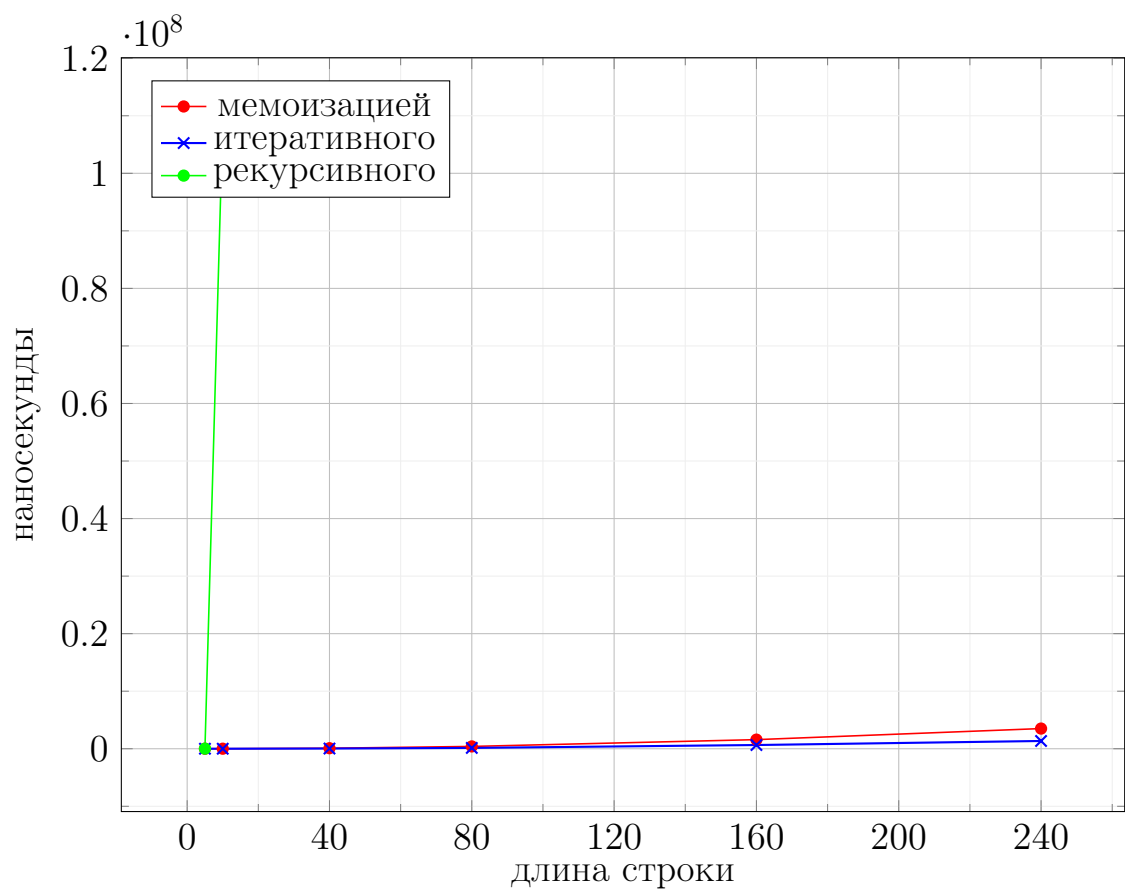


Рисунок 4.2 – Сравнение рекурсивного с мемоизацией, итеративного и рекурсивного расстояния Дамерау – Левенштейна

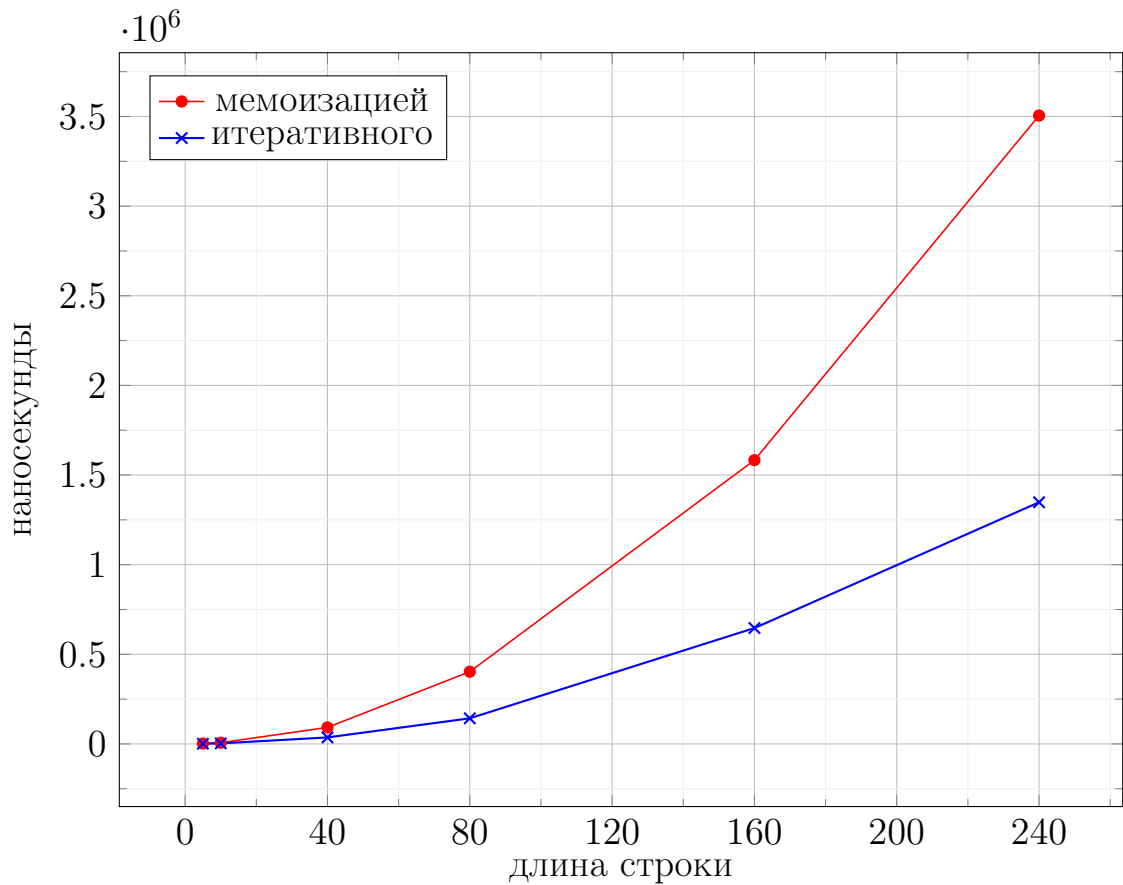


Рисунок 4.3 – Сравнение рекурсивного с мемоизацией, итеративного расстояния Дамерау – Левенштейна

4.4 Использование памяти

Далее будем считать, что $\text{sizeof}(\text{int}) = 8$, $\text{sizeof}(\text{char}) = 1$, $\text{sizeof}(\text{slice}) = 24$. Все значения в этом разделе указываются в байтах.

Длину строки S_1 обозначим как n , а длину строки S_2 — как m . Тогда затраченную память можно вычислить следующим образом:

4.4.1 Нерекурсивный алгоритм поиска расстояния Дамерау-Левенштейна

Размер выделяемой памяти:

- размер строк S_1, S_2 — $m + n$;

- размер $n = 2$ и $m = 8$;
- размер матрицы — $8 \cdot (m + 1) \cdot (n + 1)$;
- размер вспомогательных переменных в циклах — $8 + 8 + 6 \cdot 8$.

Таким образом, общая затраченная память в нерекурсивном алгоритме равняется $8mn + 9m + 9n + 88$

4.4.2 Рекурсивный алгоритм поиска расстояния без кэша Дамерау-Левенштейна

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящий строк.

Размер общей выделяемой памяти:

- размер строк $S_1, S_2 — m + n$;
- размер $n = 2$ и $m = 8$.

Размер выделяемой памяти для каждого вызова функции:

- размер аргументов функции — $2 \cdot 24 + 2 \cdot 8$;
- размер вспомогательных переменных — $6 \cdot 8$;
- размер адреса возврата — 4.

Таким образом, общая затраченная память в рекурсивном алгоритме равняется $m + n + 2 \cdot 8 + (2 \cdot 24 + 2 \cdot 8 + 6 \cdot 8 + 4) \cdot (n + m) = 117m + 117n + 16$

4.4.3 Рекурсивный алгоритм поиска расстояния с использованием кэша Дамерау-Левенштейна

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящий строк.

Размер общей выделяемой памяти:

- размер строк $S_1, S_2 - m + n$;
- размер $n = 2$ и $m = 8$;
- размер матрицы — $8 \cdot (m + 1) \cdot (n + 1)$.

Размер выделяемой памяти для каждого вызова функции:

- размер аргументов функции — $3 \cdot 24 + 2 \cdot 8$;
- размер вспомогательных переменных — $6 \cdot 8$;
- размер адреса возврата — 4.

Таким образом, общая затраченная память в рекурсивном алгоритме равняется $m + n + 2 \cdot 8 + 8 \cdot (m + 1) \cdot (n + 1) + (3 \cdot 24 + 2 \cdot 8 + 6 \cdot 8 + 4) \cdot (n + m) = 8mn + 149m + 149n + 24$

4.5 Вывод

В данном разделе были сравнены алгоритмы по памяти и по времени. Рекурсивный алгоритм Дамерау – 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. По расходу памяти все реализации проигрывают рекурсивной за счет большого количества выделенной памяти под матрицу расстояний.

То есть самым эффективным по памяти: рекурсивный алгоритм. Самый эффективный по времени: итеративный алгоритм (исходя из сделанных тестов.)

Стоит отметить, что для языков, где возможна передача указателя на массивы, самым эффективным и по времени, и по памяти будет алгоритм, использующий мемоизацию.

Заключение

В рамках лабораторной работы были:

- рассмотрены три алгоритма нахождения редакторского расстояния Дameraу – Левенштейна и одно Левенштейна;
- в аналитическом разделе были изучены смысловые различия между алгоритмами и их формульное представление;
- в рамках конструкторского раздела были получены схемы алгоритмов;
- в технологическом разделе был выбран язык программирования и представлена реализация на нем, также были приведены тестовые данные;
- в исследовательской части были сравнены алгоритмы по скорости и по памяти. Самым эффективным по времени оказался итеративный алгоритм. Самым эффективным по памяти — рекурсивный алгоритм.

По итогу реализации алгоритма поиска редакторского расстояния Дameraу – Левенштейна итеративным способом оказался быстрее остальных алгоритмов на 38 % при длине слова в 240 символов, то есть на 0.2 секунды.

В ходе лабораторной работы получены навыки динамического программирования, реализованы и изученные алгоритмы нахождения редакторского расстояния. Цель работы достигнута.

Список использованных источников

- [1] Левенштейн В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов. – М.: «Наука», Доклады АН СССР, 1965. Т. 163. С. 845–848.
- [2] Черненко В. М. Гапанюк Ю. Е. Методика идентификации пассажира по установочным данным. – М.: Вестник МГТУ им. Н.Э. Баумана. Сер. “Приборостроение”, 2012. Т. 163. С. 30–34.
- [3] Golang Документация [Электронный ресурс]. Режим доступа: <https://go.dev/doc/> (дата обращения: 24.09.2022).
- [4] Go rune. Режим доступа: <https://golangdocs.com/rune-in-golang> (дата обращения: 04.09.2022).
- [5] Pop OS 22.04 LTS [Электронный ресурс]. Режим доступа: <https://pop.system76.com> (дата обращения: 04.09.2022).
- [6] Linux – Документация [Электронный ресурс]. Режим доступа: <https://docs.kernel.org> (дата обращения: 24.09.2022).
- [7] Процессор AMD® Ryzen 7 2700 eight-core processor × 16 [Электронный ресурс]. Режим доступа: <https://www.amd.com/en/products/cpu/amd-ryzen-7-2700> (дата обращения: 04.09.2022).