



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

Введение	3
1 Аналитический раздел	4
1.1 Расстояние Левенштейна	4
1.2 Расстояние Дамерау – Левенштейна	5
1.3 Рекурсивная формула	5
1.4 Матрица расстояний	6
1.5 Рекурсивный алгоритм расстояния Дамерау – Левенштейна с мемоизацией	7
1.6 Вывод	7
2 Конструкторский раздел	9
2.1 Матричные итерационные алгоритмы	9
2.2 Модификация матричных алгоритмов	9
2.3 Рекурсивные алгоритмы	9
2.4 Вывод	14
3 Технологический раздел	15
3.1 Требования к ПО	15
3.2 Средства реализации	15
3.3 Листинги кода	15
3.3.1 Реализация алгоритмов	15
3.3.2 Подпрограммы	19
3.4 Тестовые данные	20
3.5 Вывод	20
4 Исследовательская часть	21
4.1 Технические характеристики	21
4.2 Время выполнения алгоритмов	21
4.3 Использование памяти	23
4.4 Вывод	24

Заключение	25
Список литературы	26

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 [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.

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.

В данной лабораторной работе будут рассмотрены разные реализации данных алгоритмов нахождения редакторских расстояний. Такие как: итеративный, рекурсивный и рекурсивный с кэшем.

Также будут приведены сравнения реализации по времени и памяти.

1 Аналитический раздел

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, $\alpha(u, u) = 0$ 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 $s1$ 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:

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

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:

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

$f_{X,Y}$ — 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 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 2.1 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 2.3 изображена схема алгоритма, использующая этот прием.

2.3 Рекурсивные алгоритмы

На рисунке 2.4 изображена схема рекурсивного алгоритма нахождения расстояния Дамерау – Левенштейна.

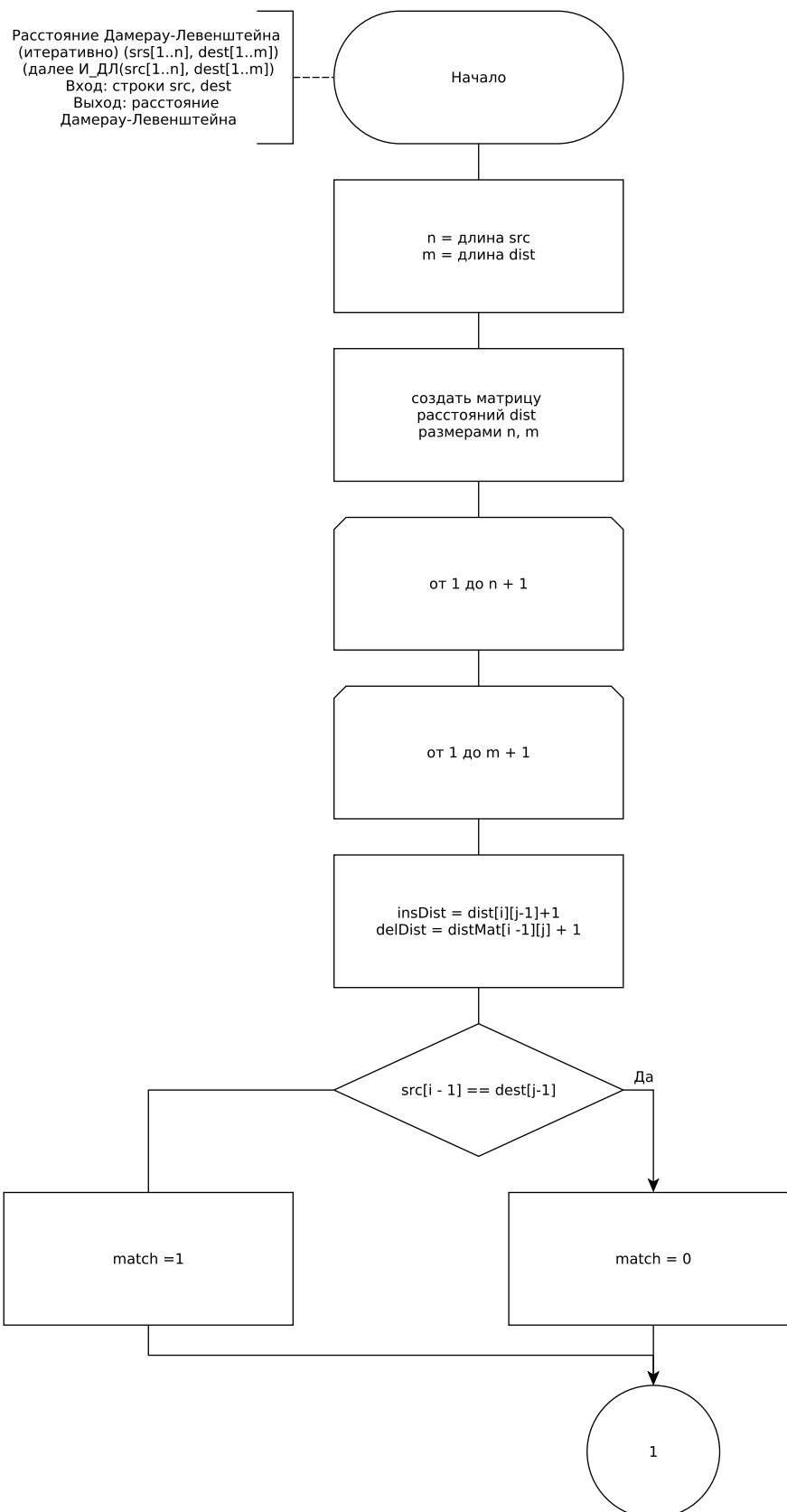


Рисунок 2.1 – Схема итерационного алгоритма расстояния Дамерау – Левенштейна с заполнением матрицы расстояний

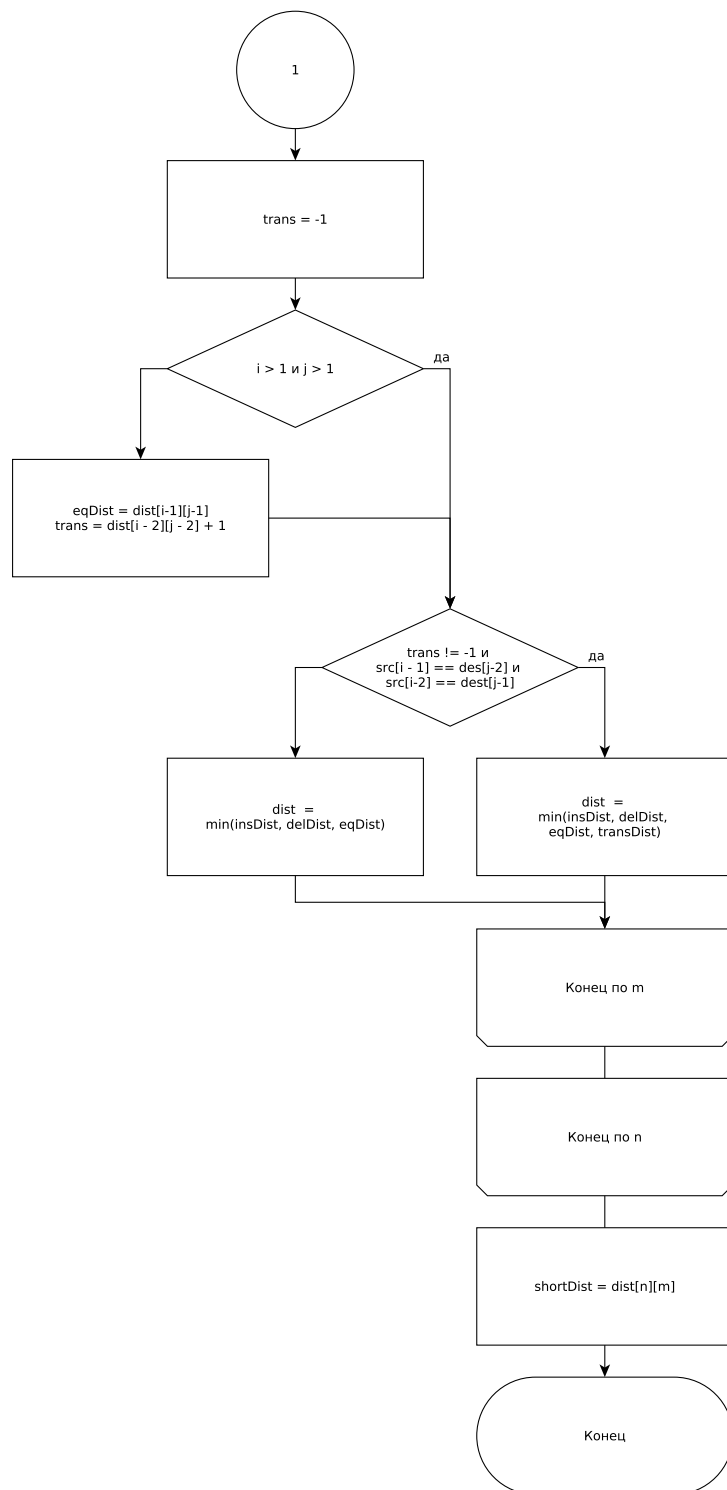


Рисунок 2.2 – Схема итерационного алгоритма расстояния Дамерау – Левенштейна с заполнением матрицы расстояний

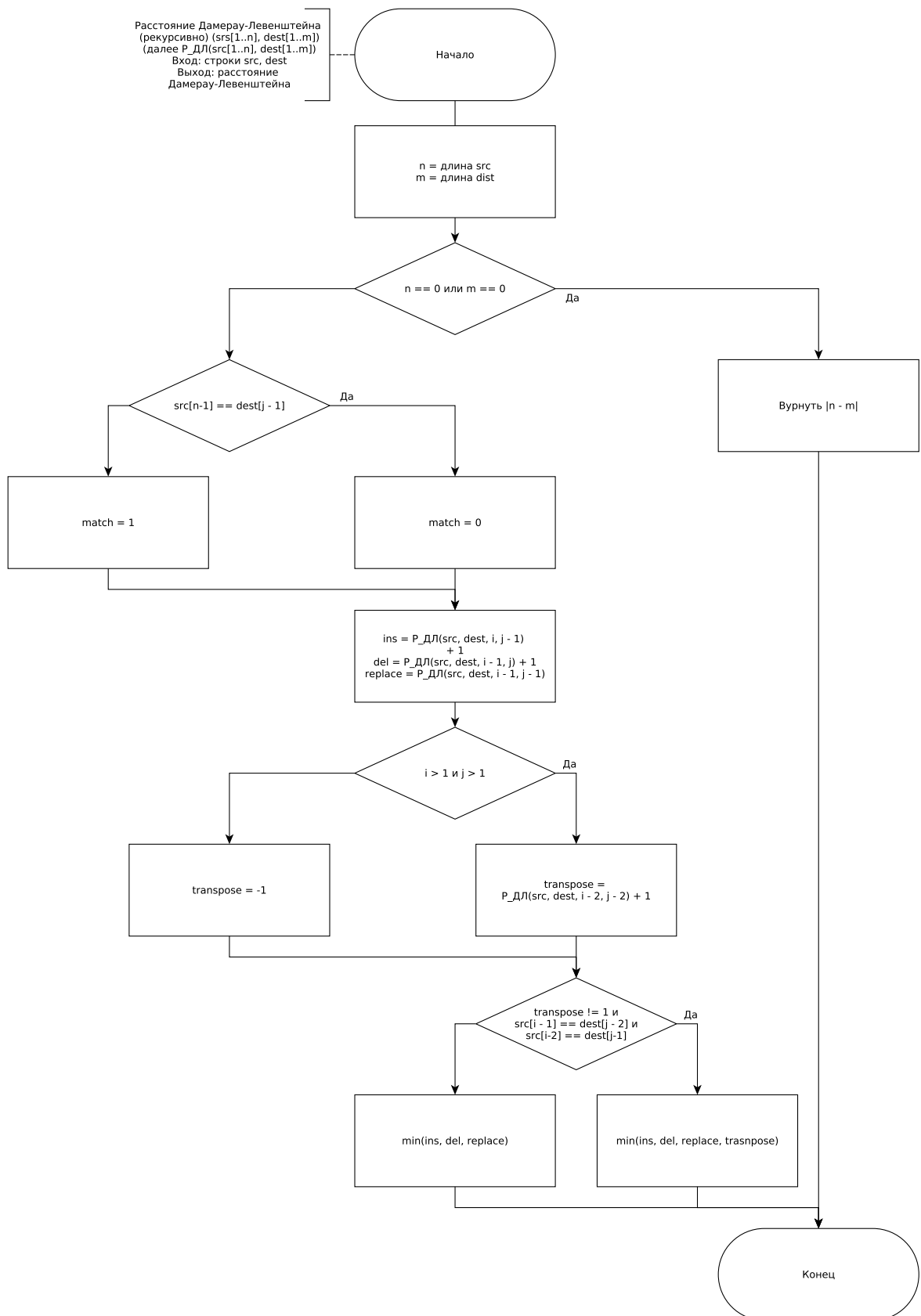


Рисунок 2.3 – Схема рекурсивного алгоритма расстояния Дамерау–Левенштейна с мемоизацией

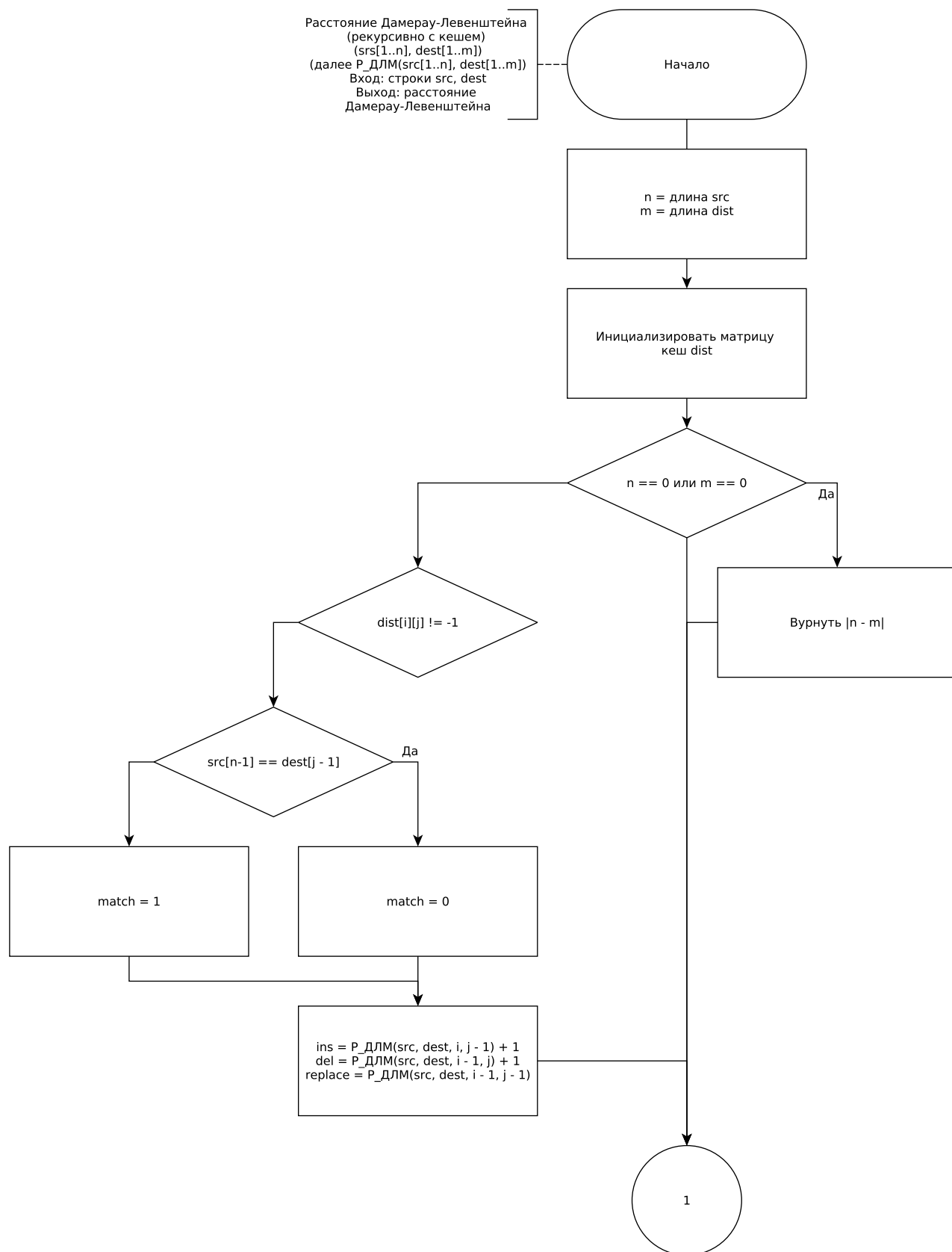


Рисунок 2.4 – Схема рекурсивного алгоритма расстояния
Дамерау – Левенштейна

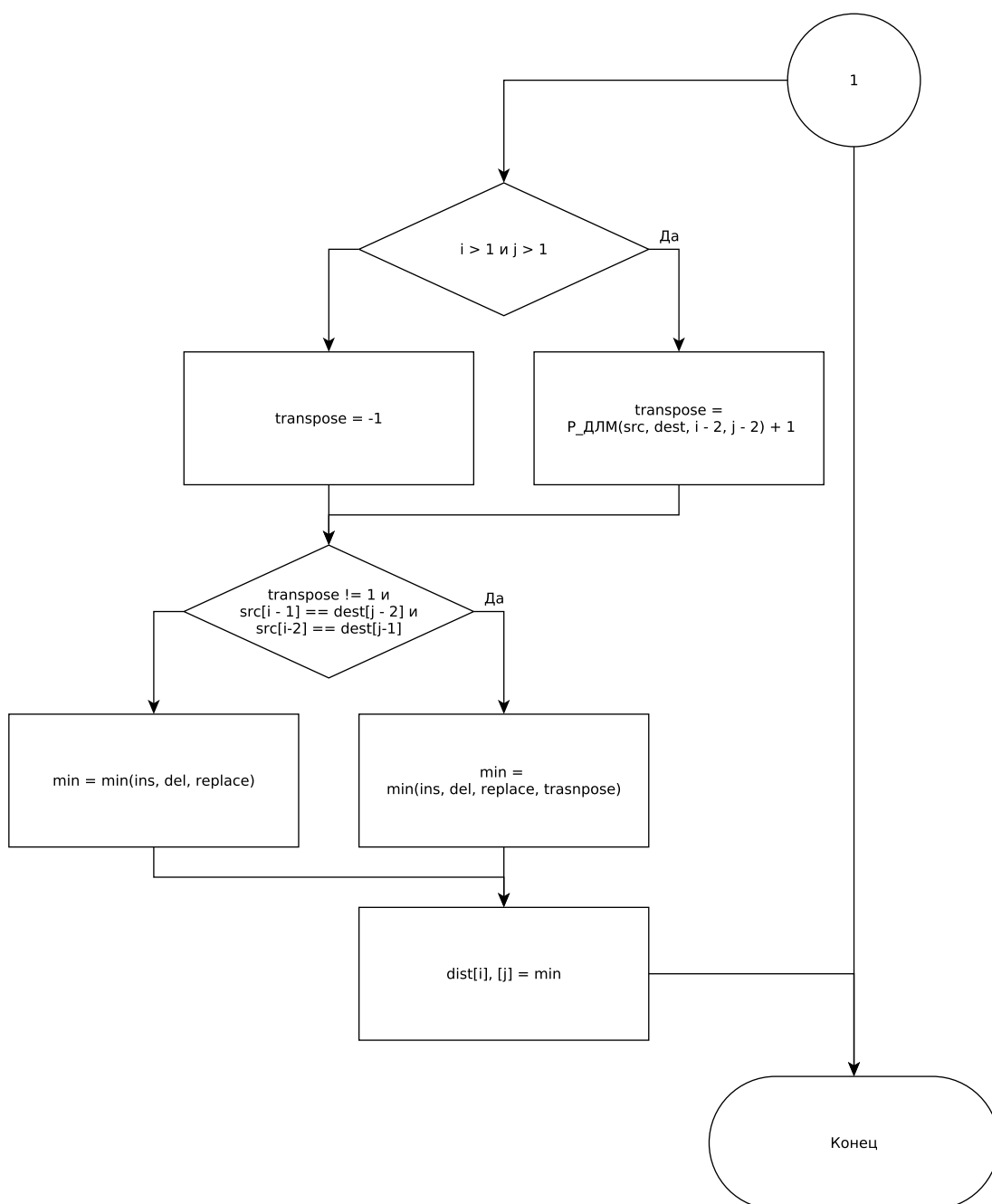


Рисунок 2.5 – Схема рекурсивного алгоритма расстояния Дамерау – Левенштейна

2.4 Вывод

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

3 Технологический раздел

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]. Данный язык предоставляет следующие возможности:

- Средства объектно-ориентированного программирования ограничиваются интерфейсами, что позволяет создавать абстракции, которые нужны для представления матрицы расстояний;
- Поддерживает библиотеки C и C++, что позволит измерить процессорное время, также имеет свою обширную библиотеку.

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

3.3 Листинги кода

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

```

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

В таблице 4 алгоритма для поиска редакторского расстояния:

- DLIter — Дамерау–Левенштейн итеративный;
- DLRec — Дамерау–Левенштейн рекурсивный;
- DLRecCache — Дамерау–Левенштейн рекурсивный с мемоизацией;
- Lev - Левенштейн итеративный.

№	S_1	S_2	DLIter	DLRec	DLRecCache	Lev
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

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 GiB;
- 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 Время выполнения алгоритмов

Алгоритмы тестировались при помощи профилирования – сбора характеристик работы программы: времени выполнения и затрат по памяти. Для каждой функции были написаны тесты оценки эффективности "бенчмарки"[8], представленные встроенными в Golang средствами. Тесты эффективности, реализованные стандартными средствами Golang автоматически делают некоторое количество замеров, предоставляя результат с некоторой погрешностью.

Листинг 4.1 – Пример теста эффективности

```
1 func BenchmarkCountDamNoRec10(b testing.B) src := "abaoboaobj"dest :=  
   "da;ldfjalj"for i := 0; i < b.N; i++ CountDamNoRec(src, dest)
```

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

В таблице 4 алгоритма для поиска редакторского расстояния:

- DLIter — Дамерау – Левенштейн итеративный;
- DLRec — Дамерау – Левенштейн рекурсивный;

- DLRecCache — Дамерау – Левенштейн рекурсивный с мемоизацией;
- Lev - Левенштейн итеративный.

Таблица 4.1 – Время выполнения алгоритмов

Длина строк	Время выполнения()			
	DRecMem	DLIter	Lev	DLRec
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	-

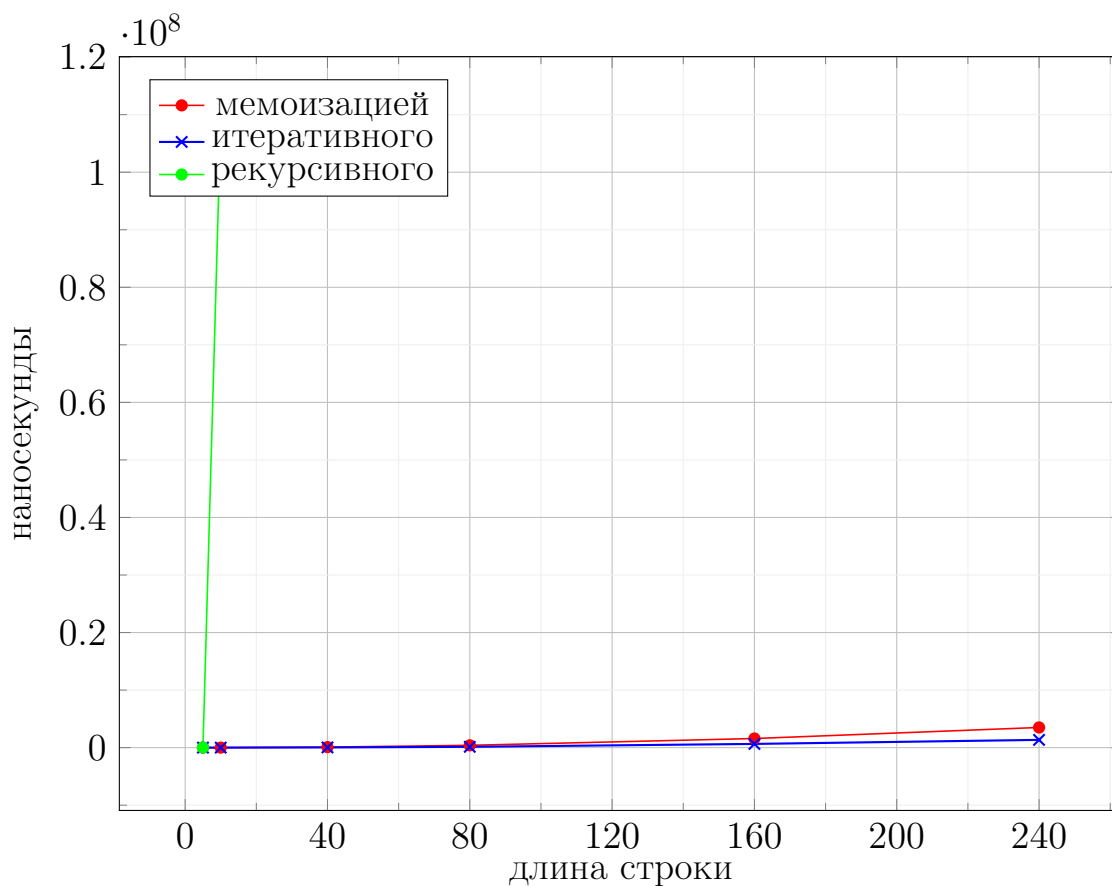


Рисунок 4.1 – Сравнение рекурсивного с мемоизацией, итеративного и рекурсивного расстояния Дамерау – Левенштейна

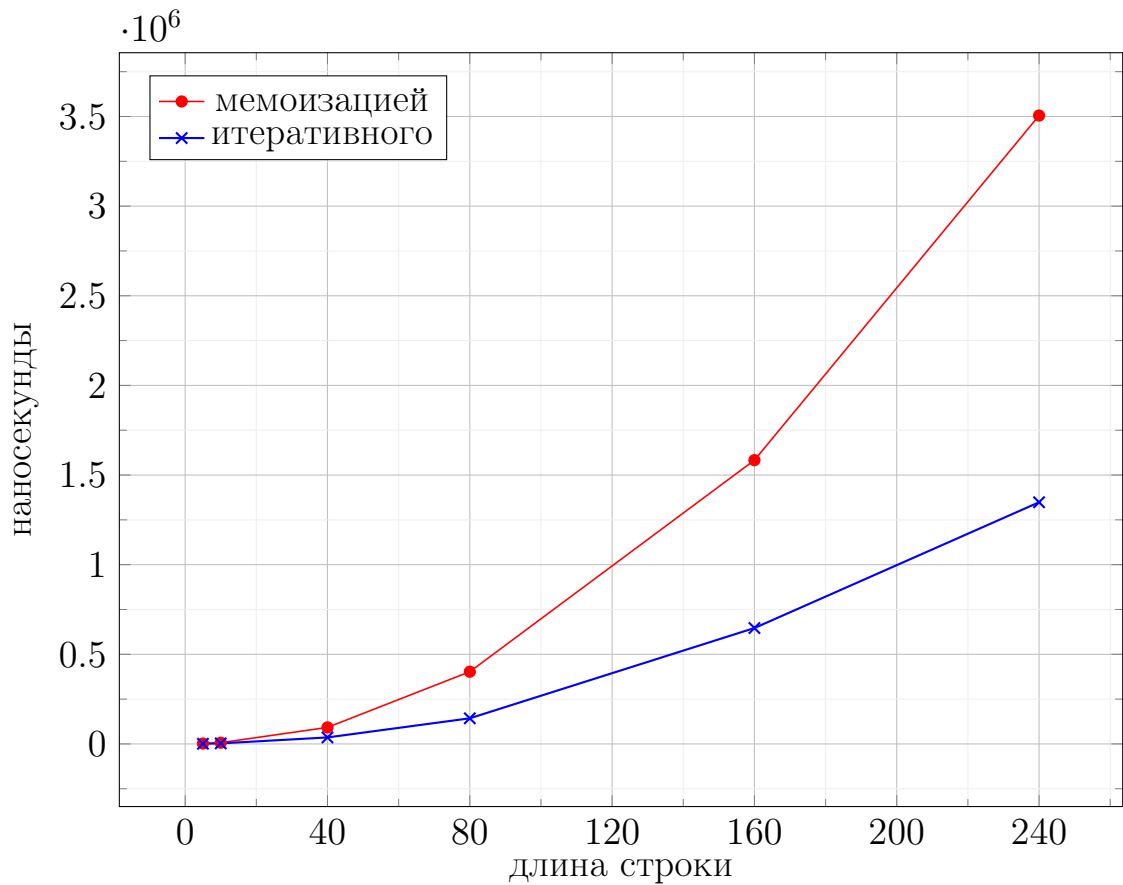


Рисунок 4.2 – Сравнение рекурсивного с мемоизацией, итеративного расстояния Дамерау – Левенштейна

4.3 Использование памяти

Максимальная глубина стека при вызове рекурсивных функций имеет следующий вид:

$$M_{recursive} = (n \cdot lvar + ret + ret_{int}) \cdot depth \quad (4.1)$$

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:

$$M_{iter} = |S_1| + |S_2| + (|S_1| + 1 \cdot |S_2| + 1) \cdot lvar + n \cdot lvar + ret + ret_{int} \quad (4.2)$$

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

4.4 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 Дамерау – 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 алгоритм поиска редакторского расстояния Дамерау – Левенштейна итеративным способом оказался быстрее остальных алгоритмов на 38 процентов при длине слова в 240 символов.

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

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] 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).
- [8] Go 1 Release Notes. Режим доступа: <https://pkg.go.dev/testing> (дата обращения: 04.09.2022).