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»

Отчет по лабораторной работе №1 по курсу "Анализ алгоритмов"

Тема Расстояние Левенштейна и Дамерау-Левенштейна

Студент Пересторонин П.Г.

Группа ИУ7-53Б

Преподаватели Волкова Л.Л., Строганов Ю.В.

Москва — 2020 г.

Оглавление

Введение

Целью данной лабораторной работы является изучить и реализовать алгоритмы нахождения расстояний Левенштейна и Дамерау – Левенштейна.

Расстояние Левенштейна (редакционное расстояние, дистанция редактирования) — метрика, измеряющая разность между двумя последовательностями символов. Она определяется как минимальное количество односимвольных операций (а именно вставки, удаления, замены), необходимых для превращения одной последовательности символов в другую. В общем случае, операциям, используемым в этом преобразовании, можно назначить разные цены. Широко используется в теории информации и компьютерной лингвистике.

Впервые задачу поставил в 1965 году советский математик Владимир Левенштейн при изучении последовательностей 0–1, впоследствии более общую задачу для произвольного алфавита связали с его именем.

Расстояние Левенштейна и его обобщения активно применяются:

1) для исправления ошибок в слове (в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи);

2) для сравнения текстовых файлов утилитой `diff` и ей подобными (здесь роль «символов» играют строки, а роль «строк» — файлы);

3) в биоинформатике для сравнения генов, хромосом и белков.

Расстояние Дамерау — Левенштейна (названо в честь учёных Фредерика Дамерау и Владимира Левенштейна) — это мера разницы двух строк символов, определяемая как минимальное количество операций вставки, удаления, замены и транспозиции (перестановки двух соседних символов), необходимых для перевода одной строки в другую. Является модификацией расстояния Левенштейна: к операциям вставки, удаления и замены символов, определённых в расстоянии Левенштейна добавлена операция транспозиции (перестановки) символов.

Задачи лабораторной работы:

- Изучение алгоритмов Левенштейна и Дамерау–Левенштейна.
- Применение методов динамического программирования для реализации алгоритмов.
- Получение практических навыков реализации алгоритмов Левенштейна и Дамерау — Левенштейна.
- Сравнительный анализ алгоритмов на основе экспериментальных данных.
- Подготовка отчета по лабораторной работе.

1 Аналитическая часть

Расстояние Левенштейна [1] между двумя строками — это минимальное количество операций вставки, удаления и замены, необходимых для превращения одной строки в другую.

Цены операций могут зависеть от вида операции (вставка, удаление, замена) и/или от участвующих в ней символов, отражая разную вероятность разных ошибок при вводе текста, и т. п. В общем случае:

- $w(a, b)$ — цена замены символа a на символ b.
- $w(\lambda, b)$ — цена вставки символа b.
- $w(a, \lambda)$ — цена удаления символа a.

Для решения задачи о редакционном расстоянии необходимо найти последовательность замен, минимизирующую суммарную цену. Расстояние Левенштейна является частным случаем этой задачи при

- $w(a, a) = 0$.
- $w(a, b) = 1,\ a \neq b$.
- $w(\lambda, b) = 1$.
- $w(a, \lambda) = 1$.

1.1 Рекурсивный алгоритм нахождения расстояния Левенштейна

Расстояние Левенштейна между двумя строками a и b может быть вычислено по формуле 1.1, где $|a|$ означает длину строки a; $a[i]$ — i-ый символ строки a , функция $D(i,j)$ определена как:

$$D(i,j) = \begin{cases} 0 & \text{i} = 0, \text{j} = 0 \\ i & \text{j} = 0, \text{i} > 0 \\ j & \text{i} = 0, \text{j} > 0 \\ \min\{ & \\ \quad D(i,j-1)+1 & \\ \quad D(i-1,j)+1 & \text{i} > 0, \text{j} > 0 \\ \quad D(i-1,j-1)+m(a[i],b[j]) & (1.2) \\ \} & \end{cases}, \qquad (1.1)$$

а функция 1.2 определена как:

$$m(a,b) = \begin{cases} 0 & \text{если a} = \text{b}, \\ 1 & \text{иначе} \end{cases}. \qquad (1.2)$$

Рекурсивный алгоритм реализует формулу 1.1. Функция D составлена из следующих соображений:

1) для перевода из пустой строки в пустую требуется ноль операций;

2) для перевода из пустой строки в строку a требуется $|a|$ операций;

3) для перевода из строки a в пустую требуется $|a|$ операций;

Для перевода из строки a в строку b требуется выполнить последовательно некоторое количество операций (удаление, вставка, замена) в некоторой последовательности. Полагая, что a', b' — строки a и b без последнего символа соответственно, цена преобразования из строки a в строку b может быть выражена как:

1) сумма цены преобразования строки a' в b и цены проведения операции удаления, которая необходима для преобразования a' в a;

2) сумма цены преобразования строки a в b' и цены проведения операции вставки, которая необходима для преобразования b' в b;

3) сумма цены преобразования из a' в b' и операции замены, предполагая, что a и b оканчиваются на разные символы;

4) цена преобразования из a' в b', предполагая, что a и b оканчиваются на один и тот же символ.

Минимальной ценой преобразования будет минимальное значение приведенных вариантов.

1.2 Матричный алгоритм нахождения расстояния Левенштейна

Прямая реализация формулы 1.1 может быть малоэффективна по времени исполнения при больших i, j, т. к. множество промежуточных значений $D(i, j)$ вычисляются заново множество раз подряд. Для оптимизации нахождения расстояния Левенштейна можно использовать матрицу в целях хранения соответствующих промежуточных значений. В таком случае алгоритм представляет собой построчное заполнение матрицы $A_{|a|\times|b|}$ значениями $D(i, j)$.

1.3 Рекурсивный алгоритм нахождения расстояния Левенштейна с заполнением матрицы

Рекурсивный алгоритм заполнения можно оптимизировать по времени выполнения с использованием матричного алгоритма. Суть данного метода заключается в параллельном заполнении матрицы при выполнении рекурсии. В случае, если рекурсивный алгоритм выполняет прогон для данных,

которые еще не были обработаны, результат нахождения расстояния заносится в матрицу. В случае, если обработанные ранее данные встречаются снова, для них расстояние не находится и алгоритм переходит к следующему шагу.

1.4 Расстояния Дамерау — Левенштейна

Расстояние Дамерау — Левенштейна может быть найдено по формуле 1.3, которая задана как

$$d_{a,b}(i,j) = \begin{cases} \max(i,j), & \text{если } \min(i,j) = 0, \\ \min\{ & \\ \quad d_{a,b}(i,j-1)+1, & \\ \quad d_{a,b}(i-1,j)+1, & \\ \quad d_{a,b}(i-1,j-1) + m(a[i],b[j]), & \text{иначе} \\ \quad \begin{cases} d_{a,b}(i-2,j-2)+1, & \text{если } i,j>1; \\ & a[i] = b[j-1]; \\ & b[j] = a[i-1] \\ \infty, & \text{иначе} \end{cases} & \\ \} & \end{cases}, \tag{1.3}$$

Формула выводится по тем же соображениям, что и формула (1.1). Как и в случае с рекурсивным методом, прямое применение этой формулы неэффективно по времени исполнения, поэтому аналогично методу из 1.3 производится добавление матрицы для хранения промежуточных значений рекурсивной формулы.

Вывод

Формулы Левенштейна и Дамерау — Левенштейна для рассчета расстояния между строками задаются рекурсивно, а следовательно, алгоритмы

могут быть реализованы рекурсивно или итерационно.

Пусть S_1 и S_2 — две строки (длиной M и N соответственно) над некоторым алфавитом, тогда расстояние Левенштейна $d(S_1, S_2)$ можно подсчитать по рекуррентной формуле $d(S_1, S_2) = D(M, N)$, где

$$D(i,j) = \begin{cases} 0, & i = 0,\ j = 0 \\ i, & i > 0,\ j = 0 \\ j, & i = 0,\ j > 0 \\ \min\{ & \\ \quad D(i, j-1) + 1, & \\ \quad D(i-1, j) + 1, & i > 0,\ j > 0 \\ \quad D(i-1, j-1) + m(S_1[i], S_2[j]) & \\ \} & \end{cases} \tag{1.4}$$

2 Конструкторская часть

2.1 Схема алгоритма нахождения расстояния Левенштейна

На рисунке 2.1 приведена схема рекурсивного алгоритма нахождения расстояния Левенштейна.

На рисунке 2.2 схема алгоритма инициализации матрицы.

На рисунках 2.3 и 2.4 приведена схема рекурсивного алгоритма нахождения расстояния Левенштейна с заполнением матрицы.

На рисунке 2.5 приведена схема алгоритма нахождения расстояния расстояния Левенштейна с заполнением матрицы.

2.2 Схема алгоритма нахождения расстояния Дамерау — Левенштейна

На рисунках 2.6 и 2.7 приведена схема матричного алгоритма нахождения расстояния Дамерау – Левенштейна.

Вывод

На основе теоретических данных, полученных из аналитического раздела были построены схемы требуемых алгоритмов.

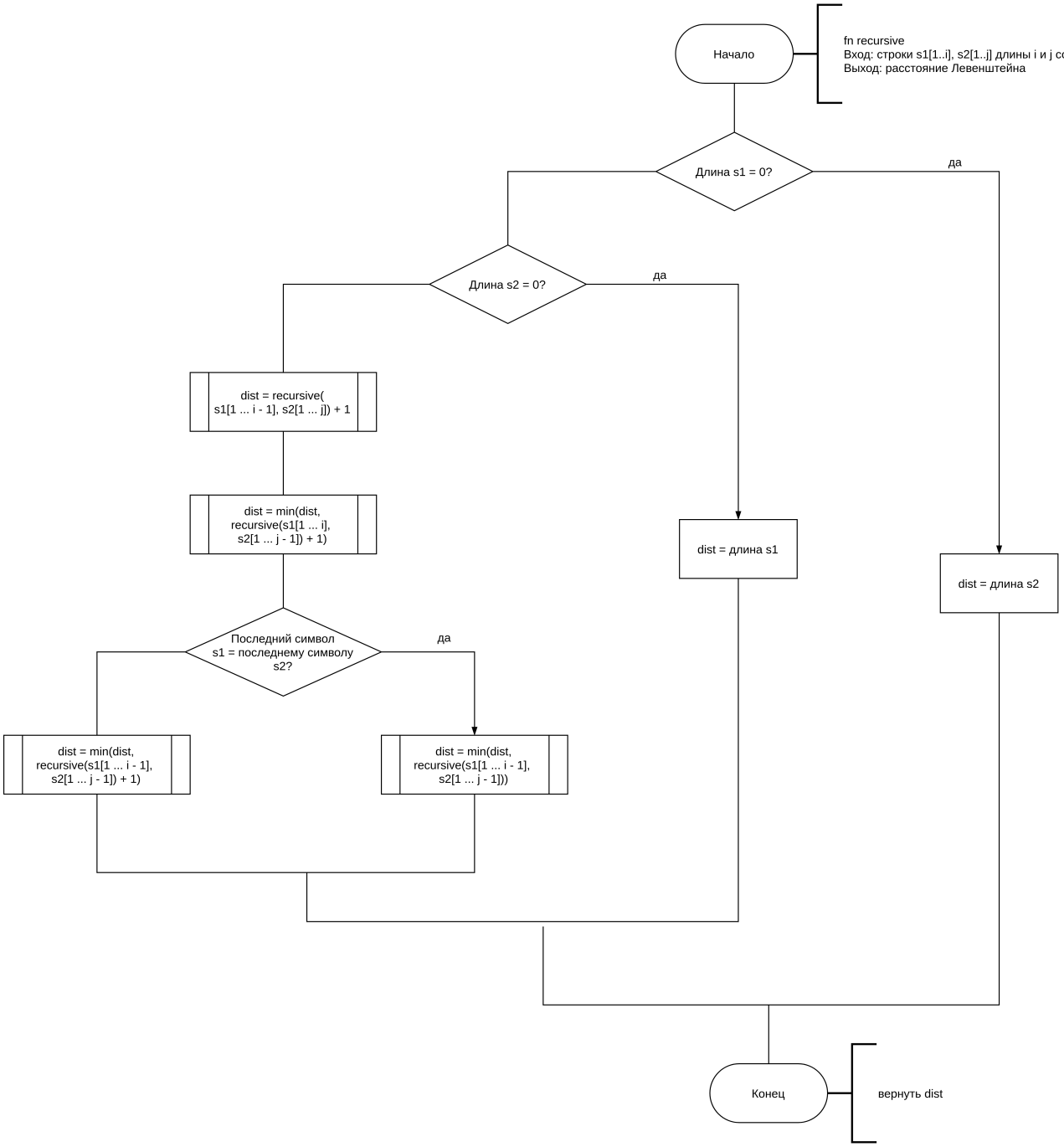

Рисунок 2.1 – Схема рекурсивного алгоритма нахождения расстояния Левенштейна

Рисунок 2.2 – Схема алгоритма инициализации матрицы

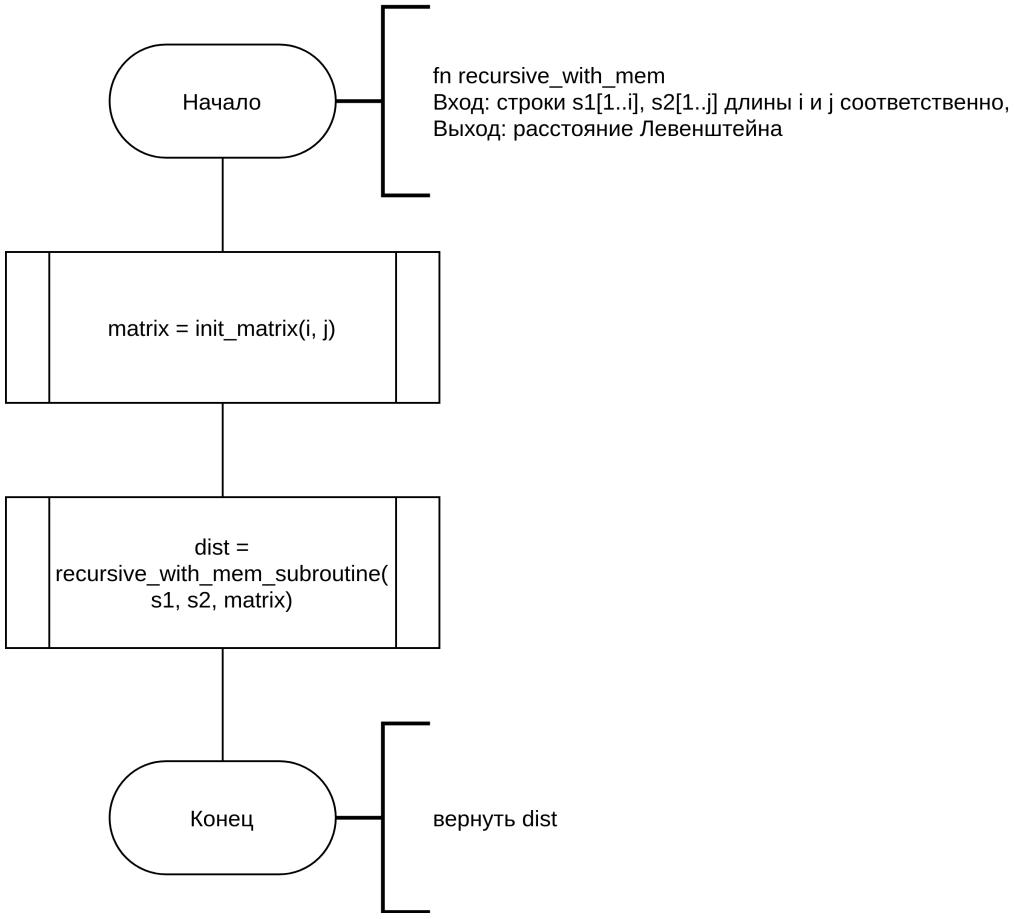

Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Левенштейна с заполнением матрицы

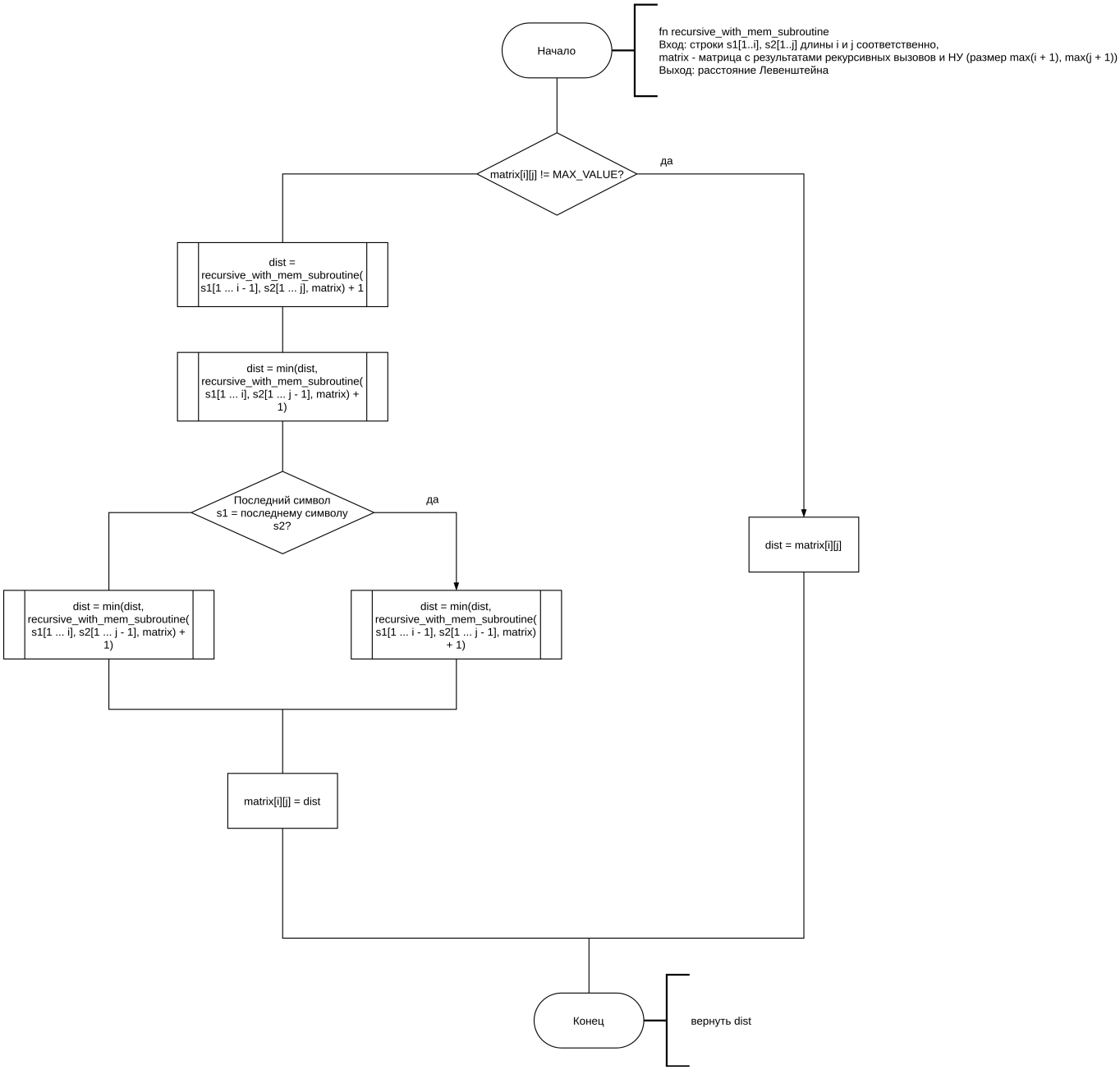

Рисунок 2.4 – Схема процедуры рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с заполнением матрицы

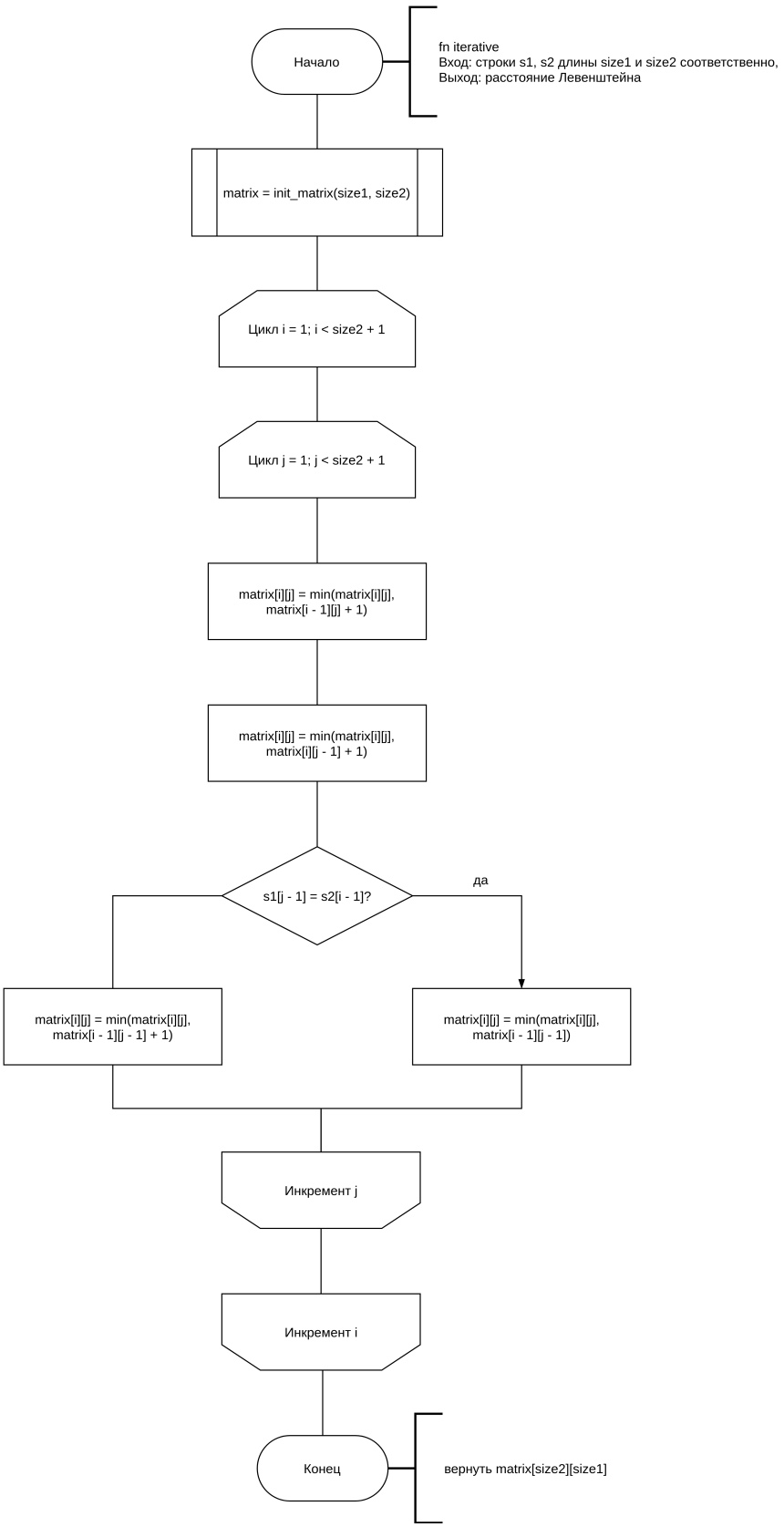

Рисунок 2.5 – Схема матричного алгоритма нахождения расстояния Левенштейна

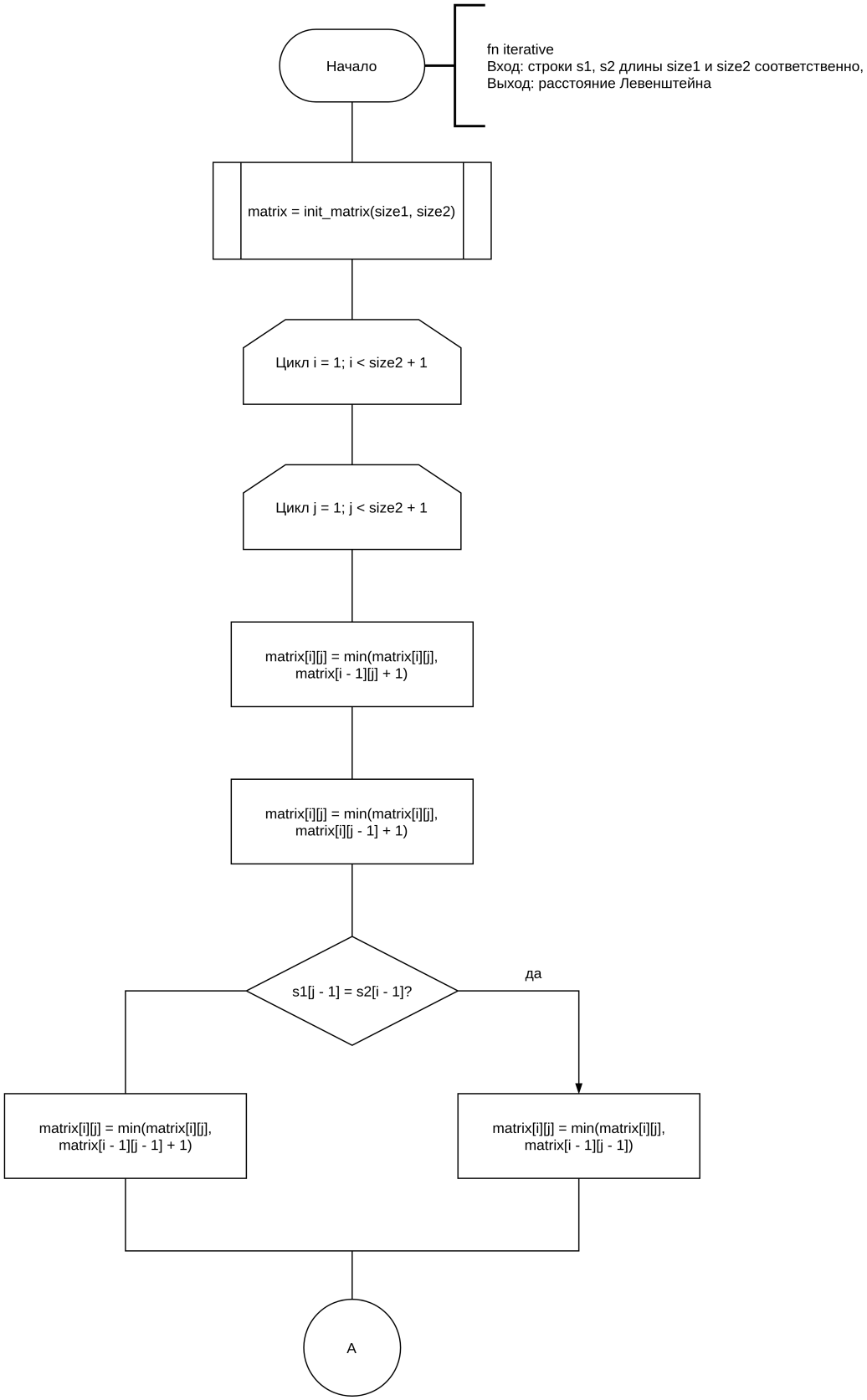

Рисунок 2.6 – Схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна

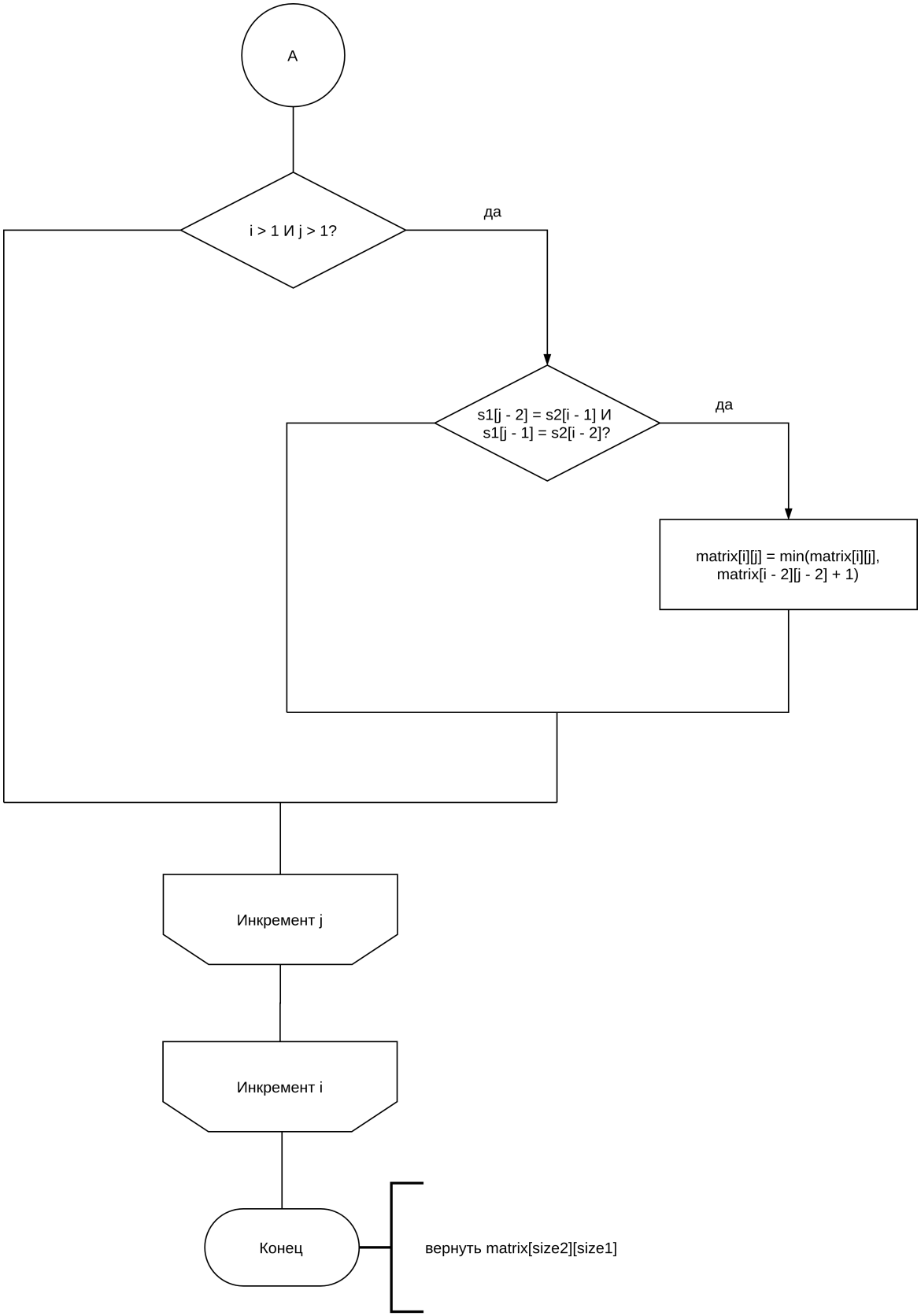

Рисунок 2.7 – Продолжение схемы матричного алгоритма нахождения расстояния Дамерау-Левенштейна

3 Технологическая часть

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

3.1 Требования к ПО

К программе предъявляется ряд требований:

- на вход подаются две строки на русском или английском языке в любом регистре;
- на выходе — искомое расстояние для всех четырех методов и матрицы расстояний для всех методов, за исключением рекурсивного.

3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран современный компилируемый ЯП Rust [2]. Данный выбор обусловлен моим желанием расширить свои знания в области применения данного языка.

3.3 Листинг кода

В листинге 3.1 приведена реализация алгоритмов Левенштейна и Дамерау — Левенштейна, а также вспомогательные функции.

Листинг 3.1 – Листинг с алгоритмами

```
use std::cmp::{min, max};

pub fn recursive(s1: &[char], s2: &[char]) -> (usize, usize) {
    _recursive(s1, s2, 0)
}

fn _recursive(s1: &[char], s2: &[char], depth: usize) -> (usize, usize) {
    if s1.len() == 0 {
```

```
        return (s2.len(), depth);
    } else if s2.len() == 0 {
        return (s1.len(), depth);
    }

    // insertion
    let (best_score, max_depth) = _recursive(&s1[..(s1.len() - 1)], s2, depth + 1);
    let best_score = best_score + 1;
    // deletion
    let (score, cur_depth) = _recursive(s1, &s2[..(s2.len() - 1)], depth + 1);
    let (best_score, max_depth) = (min(best_score, score + 1), max(cur_depth,
        max_depth));
    // match/replace
    let (score, cur_depth) = _recursive(&s1[..(s1.len() - 1)], &s2[..(s2.len() - 1)],
        depth + 1);
    let score = if s1[s1.len() - 1] == s2[s2.len() - 1] { score } else { score + 1 };
    let (best_score, max_depth) = (min(best_score, score), max(cur_depth, max_depth));
    (best_score, max_depth)
}

pub fn recursive_with_mem(s1: &[char], s2: &[char]) -> (usize, usize, Vec<Vec<usize>>) {
    let mut matrix = vec![vec![usize::MAX; s1.len() + 1]; s2.len() + 1];
    init_matrix(&mut matrix);
    let res = _recursive_with_mem(s1, s2, 0, &mut matrix);
    (res.0, res.1, matrix)
}

fn init_matrix(matrix: &mut [Vec<usize>]) {
    if matrix.len() == 0 {
        return;
    }

    for i in 0..matrix.len() {
        matrix[i][0] = i;
    }

    for j in 0..matrix[0].len() {
        matrix[0][j] = j;
    }
}

fn _recursive_with_mem(s1: &[char], s2: &[char], depth: usize, matrix: &mut
    [Vec<usize>]) -> (usize, usize) {
    if matrix[s2.len()][s1.len()] != usize::MAX {
        return (matrix[s2.len()][s1.len()], depth);
    }

    // insertion
```

```
54      let (best_score, max_depth) = _recursive_with_mem(&s1[..(s1.len() - 1)], &s2, depth
            + 1, matrix);
55      let best_score = best_score + 1;
56      // deletion
57      let (score, cur_depth) = _recursive_with_mem(&s1, &s2[..(s2.len() - 1)], depth + 1,
            matrix);
58      let (best_score, max_depth) = (min(best_score, score + 1), max(cur_depth,
            max_depth));
59      // match/replace
60      let (score, cur_depth) = _recursive_with_mem(&s1[..(s1.len() - 1)], &s2[..(s2.len()
            - 1)], depth + 1, matrix);
61      let score = if s1[s1.len() - 1] == s2[s2.len() - 1] { score } else { score + 1 };
62      let (best_score, max_depth) = (min(best_score, score), max(cur_depth, max_depth));
63      matrix[s2.len()][s1.len()] = best_score;
64      (best_score, max_depth)
65  }
66
67  pub fn iterative(s1: &[char], s2: &[char]) -> (usize, Vec<Vec<usize>>) {
68      let mut matrix = vec![vec![usize::MAX; s1.len() + 1]; s2.len() + 1];
69      init_matrix(&mut matrix);
70
71      for i in 1..(s2.len() + 1) {
72          for j in 1..(s1.len() + 1) {
73              // insertion
74              matrix[i][j] = min(matrix[i][j], matrix[i - 1][j] + 1);
75              // deletion
76              matrix[i][j] = min(matrix[i][j], matrix[i][j - 1] + 1);
77              // match/replace
78              let score = if s1[j - 1] == s2[i - 1] { matrix[i - 1][j - 1] } else {
                    matrix[i - 1][j - 1] + 1 };
79              matrix[i][j] = min(matrix[i][j], score);
80          }
81      }
82
83      (matrix[s2.len()][s1.len()], matrix)
84  }
85
86  pub fn iterative_dl(s1: &[char], s2: &[char]) -> (usize, Vec<Vec<usize>>) {
87      let mut matrix = vec![vec![usize::MAX; s1.len() + 1]; s2.len() + 1];
88      init_matrix(&mut matrix);
89
90      for i in 1..(s2.len() + 1) {
91          for j in 1..(s1.len() + 1) {
92              // insertion
93              matrix[i][j] = min(matrix[i][j], matrix[i - 1][j] + 1);
94              // deletion
95              matrix[i][j] = min(matrix[i][j], matrix[i][j - 1] + 1);
96              // match/replace
```

```
            let score = if s1[j - 1] == s2[i - 1] { matrix[i - 1][j - 1] } else {
                matrix[i - 1][j - 1] + 1 };
            matrix[i][j] = min(matrix[i][j], score);

            if i > 1 && j > 1 {
                let transition_condition = s1[j - 1] == s2[i - 2] && s1[j - 2] == s2[i -
                    1];
                if transition_condition {
                    matrix[i][j] = min(matrix[i][j], matrix[i - 2][j - 2] + 1);
                }
            }
        }
    }

    (matrix[s2.len()][s1.len()], matrix)
}

#[cfg(test)]
mod tests;
```

В таблице 3.1 приведены функциональные тесты для алгоритмов вычисления расстояния Левенштейна и Дамерау — Левенштейна. Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты

Строка 1	**Строка 2**	**Ожидаемый результат**	
		Левенштейн	**Дамерау — Левенштейн**
cook	cooker	2	2
mother	money	3	3
woman	water	4	4
program	friend	6	6
house	girl	5	5
probelm	problem	2	1
head	ehda	3	2
bring	brought	4	4
happy	happy	0	0
minute	moment	5	5
person	eye	5	5
week	weeks	1	1
member	morning	6	6
death	health	2	2
education	question	4	4
room	moor	2	2
car	city	3	3
air	area	3	3

Вывод

Были разработаны и протестированы алгоритмы: нахождения расстояния Левенштейна рекурсивно, с заполнением матрицы и рекурсивно с заполнением матрицы, а также нахождения расстояния Дамерау — Левенштейна с заполнением матрицы.

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- Операционная система: Manjaro [3] Linux [4] 20.1 64-bit.
- Память: 8 GiB.
- Процессор: Intel® Core™ i7-8550U[5].

Тестирование проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, окружением, а также непосредственно системой тестирования.

4.2 Время выполнения алгоритмов

Алгоритмы тестировались при помощи написания «бенчмарков» [6], предоставляемых встроенными в Rust средствами. Такие бенчмарки делают за нас некоторое кол-во размеров, предоставляя затем результат с некоторой погрешностью. Также мною были написаны тесты, прогоняющие алгоритмы Z раз, где Z можно выбирать.

В листинге 4.1 пример реализации бенчмарка.

Листинг 4.1 – Пример бенчмарка

```
#[cfg(test)]
mod benchs {
    use super::*;

    #[bench]
    fn iterative10(b: &mut Bencher) {
        let s1 = generate_string_of_size(10);
        let s2 = generate_string_of_size(10);
        b.iter(|| algorithms::iterative(&s1, &s2));
    }
}
```

Результаты замеров приведены в таблице 4.1. В данной таблице для значений, для которых тестирование не выполнялось, в поле результата находится NaN. На рисунках 4.1 и 4.2 приведены графики зависимостей времени работы алгоритмов от длины строк.

Таблица 4.1 – Замер времени для строк, размером от 5 до 200

Длина строк	**Время, нс**			
	Rec	**RecMat**	**ItMat**	**DL**
10	32766430	1313	634	681
20	NaN	5157	2367	2582
30	NaN	11342	4813	5207
50	NaN	30066	12518	13533
100	NaN	116134	48111	52078
200	NaN	529335	249057	527797

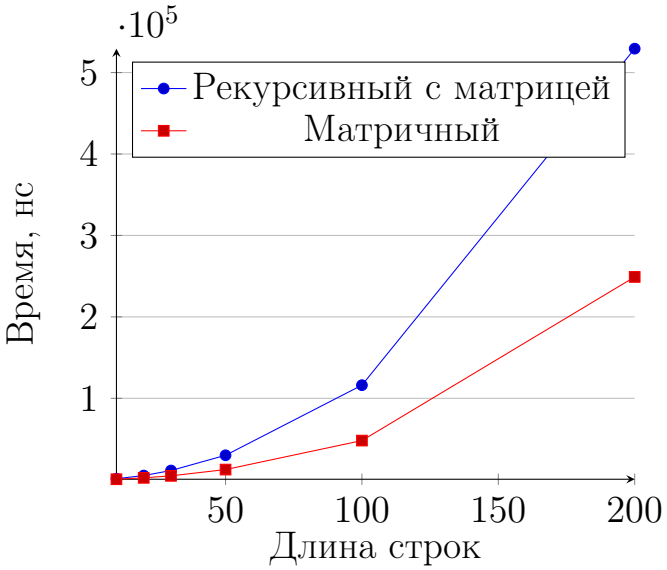

Рисунок 4.1 – Зависимость времени работы алгоритма вычисления расстояния Левенштейна от длины строк (рекурсивная с заполнением матрицы и матричная реализации)

4.3 Использование памяти

Алгоритмы Левенштейна и Дамерау — Левенштейна не отличаются друг от друга с точки зрения использования памяти, следовательно, до-

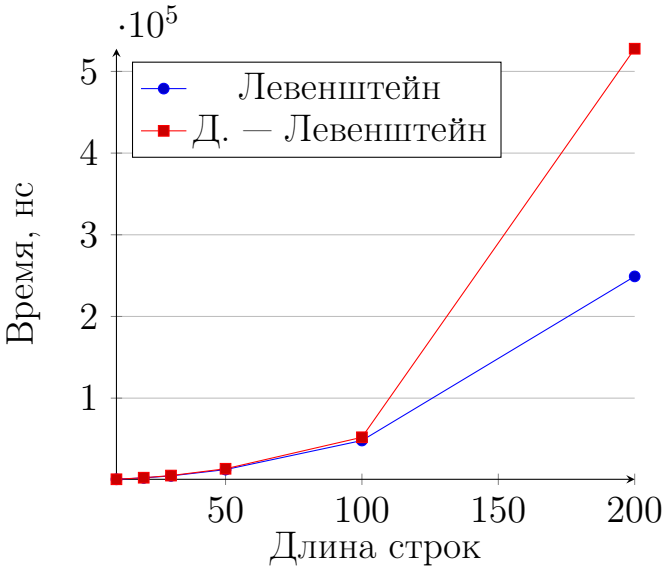

Рисунок 4.2 – Зависимость времени работы матричных реализаций алгоритмов Левенштейна и Дамерау — Левенштейна

статочно рассмотреть лишь разницу рекурсивной и матричной реализаций этих алгоритмов.

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк, соответственно, максимальный расход памяти (4.1)

$$(\mathcal{C}(S_1) + \mathcal{C}(S_2)) \cdot (2 \cdot \mathcal{C}(\text{string}) + 3 \cdot \mathcal{C}(\text{int})), \tag{4.1}$$

где $\mathcal{C}$ — оператор вычисления размера, S_1, S_2 — строки, int — целочисленный тип, string — строковый тип.

Использование памяти при итеративной реализации теоретически равно

$$(\mathcal{C}(S_1) + 1) \cdot (\mathcal{C}(S_2) + 1) \cdot \mathcal{C}(\text{int}) + 10 \cdot \mathcal{C}(\text{int}) + 2 \cdot \mathcal{C}(\text{string}). \tag{4.2}$$

Вывод

Рекурсивный алгоритм Левенштейна работает на порядок дольше итеративных реализаций, время его работы увеличивается в геометрической прогрессии. На словах длиной 10 символов, матричная реализация алгоритма Левенштейна превосходит по времени работы рекурсивную на несколько порядков. Рекурсивный алгоритм с заполнением матрицы пре-

восходит простой рекурсивный и сравним по времени работы с матричными алгоритмами. Алгоритм Дамерау — Левенштейна по времени выполнения сопоставим с алгоритмом Левенштейна. В нём добавлены дополнительные проверки, и по сути он является алгоритмом другого смыслового уровня.

Но по расходу памяти итеративные алгоритмы проигрывают рекурсивному: максимальный размер используемой памяти в них растёт как произведение длин строк, в то время как у рекурсивного алгоритма — как сумма длин строк.

Заключение

В ходе выполнения работы были выполнены все поставленные задачи и изучены методы динамического программирования на основе алгоритмов вычисления расстояния Левенштейна.

Экспериментально были установлены различия в производительности различных алгоритмов нахождения расстояния Левенштейна. Рекурсивный алгоритм Левенштейна работает на несколько порядков медленнее матричной реализации. Рекурсивный алгоритм с параллельным заполнением матрицы работает быстрее простого рекурсивного, но все еще медленнее матричного. Если длина сравниваемых строк превышает 10, рекурсивный алгоритм становится неприемлимым для использования по времени выполнения программы. Матричная реализация алгоритма Дамерау — Левенштейна сопоставима с алгоритмом Левенштейна. В ней добавлены дополнительные проверки, что делает его немного медленнее.

Теоретически было рассчитано использования памяти в каждом из алгоритмов нахождения расстояния Левенштейна. Обычные матричные алгоритмы потребляют намного больше памяти, чем рекурсивные, за счет дополнительного выделения памяти под матрицы.

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] Rust Programming Language [Электронный ресурс]. Режим доступа: `https://www.rust-lang.org/` (дата обращения: 11.09.2020).

[3] Manjaro – enjoy the simplicity [Электронный ресурс]. Режим доступа: `https://manjaro.org/` (дата обращения: 14.09.2020).

[4] Linux – Википедия [Электронный ресурс]. Режим доступа: `https://ru.wikipedia.org/wiki/Linux` (дата обращения: 14.09.2020).

[5] Процессор Intel® Core™ i7-8550U [Электронный ресурс] Режим доступа: `https://ark.intel.com/content/www/ru/ru/ark/products/122589/intel-core-i7-8550u-processor-8m-cache-up-to-4-00-ghz.html` (дата обращения: 14.09.2020).

[6] Документация по ЯП Rust: бенчмарки [Электронный ресурс]. Режим доступа: `https://doc.rust-lang.org/1.7.0/book/benchmark-tests.html` (дата обращения: 12.09.2020).