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»

Лабораторная работа №1
по дисциплине "Анализ Алгоритмов"

Тема Расстояние Левенштейна

Студент Рядинский К. В.

Группа ИУ7-53Б

Преподаватель Волкова Л. Л.

Москва

2021 г.

Оглавление

Введение

Расстояние Левенштейна - минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

Расстояние Левенштейна применяется в теории информации и компьютерной лингвистике для следущего:

- исправления ошибок в слове;
- сравнения текстовых файлов утилитой diff;
- в биоинформатике для сравнения генов, хромосом и белков.

Цели данной лабораторной работы:

1. Изучение метода динамического программирования на материале алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.
2. Оценка реализаций алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейн

Задачами данной лабораторной являются:

1. Изучение алгоритмов Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;
2. Применение метода динамического программирования для матричной реализации указанных алгоритмов;
3. Получение практических навыков реализации указанных алгоритмов: двух алгоритмов в матричной версии и одного из алгоритмов в рекурсивной версии;
4. Сравнительный анализ линейной и рекурсивной реализаций выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);
5. Экспериментальное подтверждение различий во временно́й эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк;
6. Описание и обоснование полученных результатов в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

1 | Аналитическая часть

Расстояние Левенштейна [2] между двумя строками – это минимальное количество операций вставки, удаления и замены, необходимых для превращения одной строки в другую

Цены операций могут зависеть от вида операций (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 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 это задачи при

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

1.1 Расстояние Левенштейна

Пусть S_1 и S_2 — две строки (длиной M и N соответственно) над некоторым алфавитом, тогда расстояние Левенштейна можно подсчитать по следующей рекуррентной формуле:

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

где $m(a, b)$ равна нулю, если $a = b$ и единице в противном случае; $min\{\, a, b, c\}$ возвращает наименьший из аргументов.

1.2 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна вычисляется по следующей рекуррентной формуле:

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & i > 0, j = 0 \\ j, & i = 0, j > 0 \\ min \begin{cases} D(i,j-1)+1, \\ D(i-1,j)+1, \\ D(i-1,j-1)+m(S_1[i],S_2[i]), \\ D(i-2,j-2)+m(S_1[i],S_2[i]), \end{cases} & \text{, если } i,j > 0 \\ & \text{и } S_1[i] = S_2[j-1] \\ & \text{и } S_1[i-1] = S_2[j] \\ min \begin{cases} D(i,j-1)+1, \\ D(i-1,j)+1, \\ D(i-1,j-1)+m(S_1[i],S_2[i]), \end{cases} & \text{, иначе} \end{cases}$$

1.3 Вывод

В данном разделе были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна, который является модификаций первого, учитывающего возможность перестановки соседних символов.

2 | Конструкторская часть

На рис. 2.1 приведена схема рекурсивного алгоритма Левенштейна
На рис. 2.2 приведена схема рекурсивного алгоритма Дамерау-Левенштейна
На рис. 2.5 приведена схема матричного алгоритма Левенштейна
На рис. 2.6 приведена схема матричного алгоритма Дамерау-Левенштейна

2.1 Схемы алгоритмов

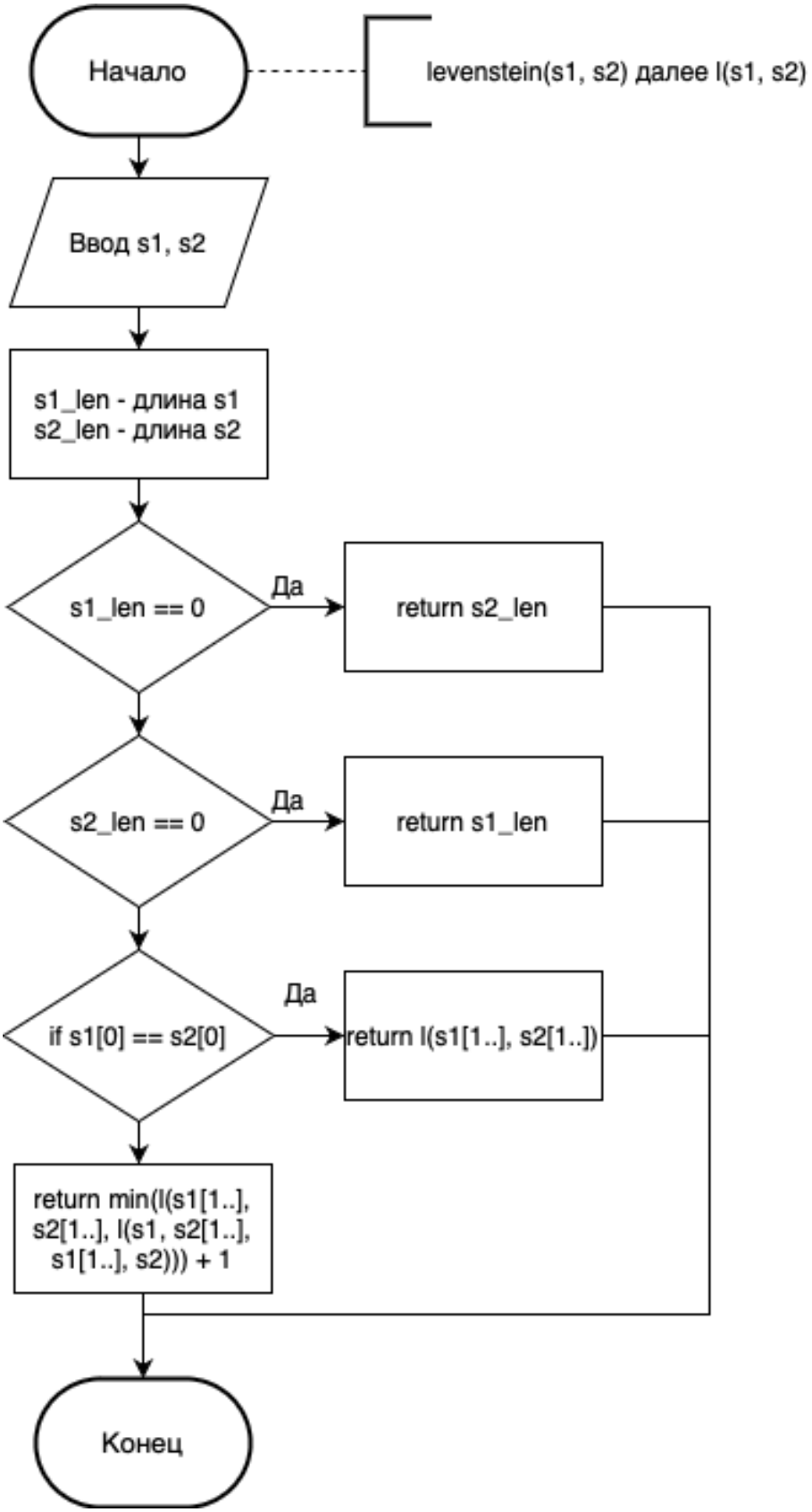

Рис. 2.1: Схема рекурсивного алгоритма нахождения расстояния Левенштейна

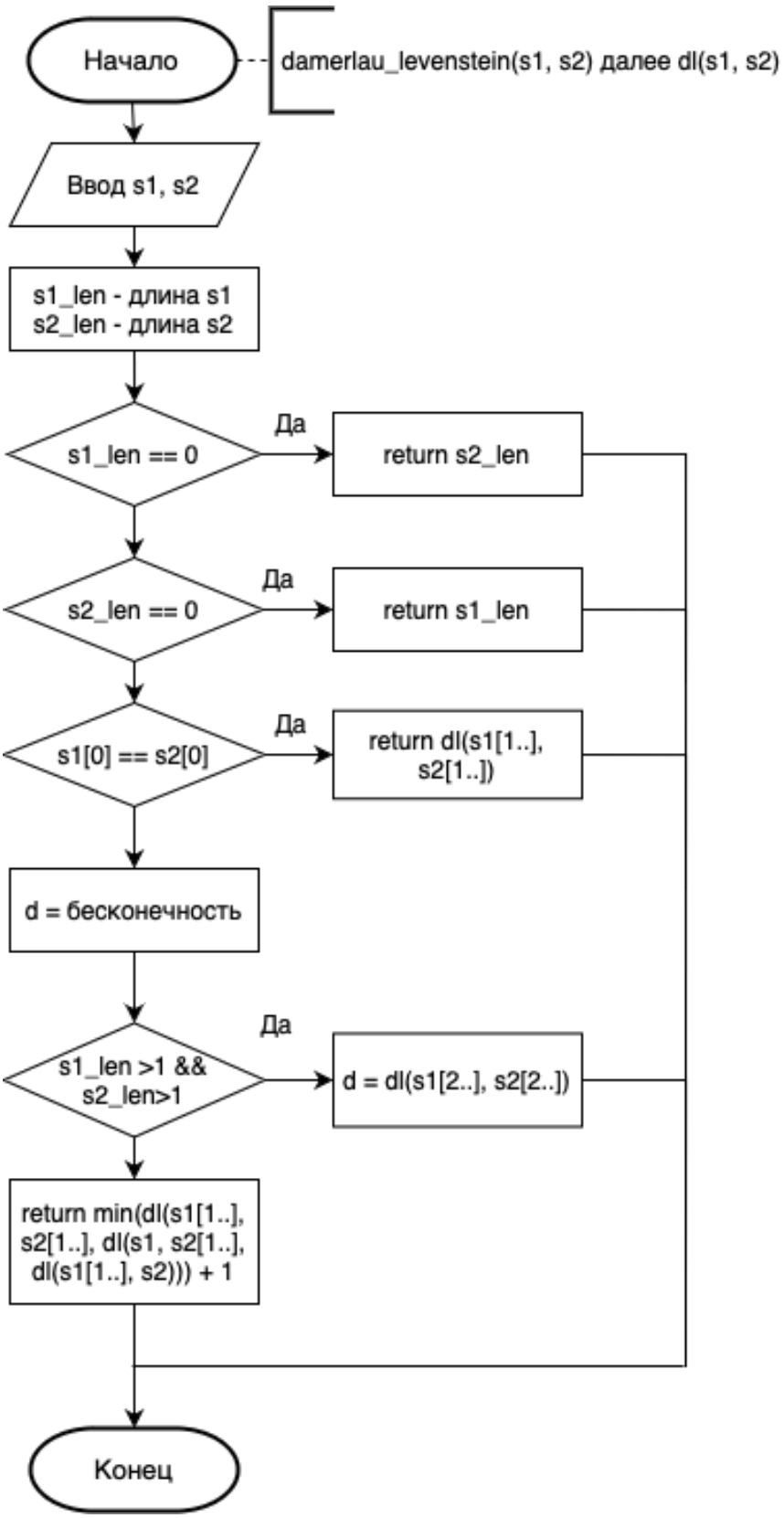

Рис. 2.2: Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

Рис. 2.3: Схема рекурсивного алгоритма нахождения расстояния Левенштейна с кэшем

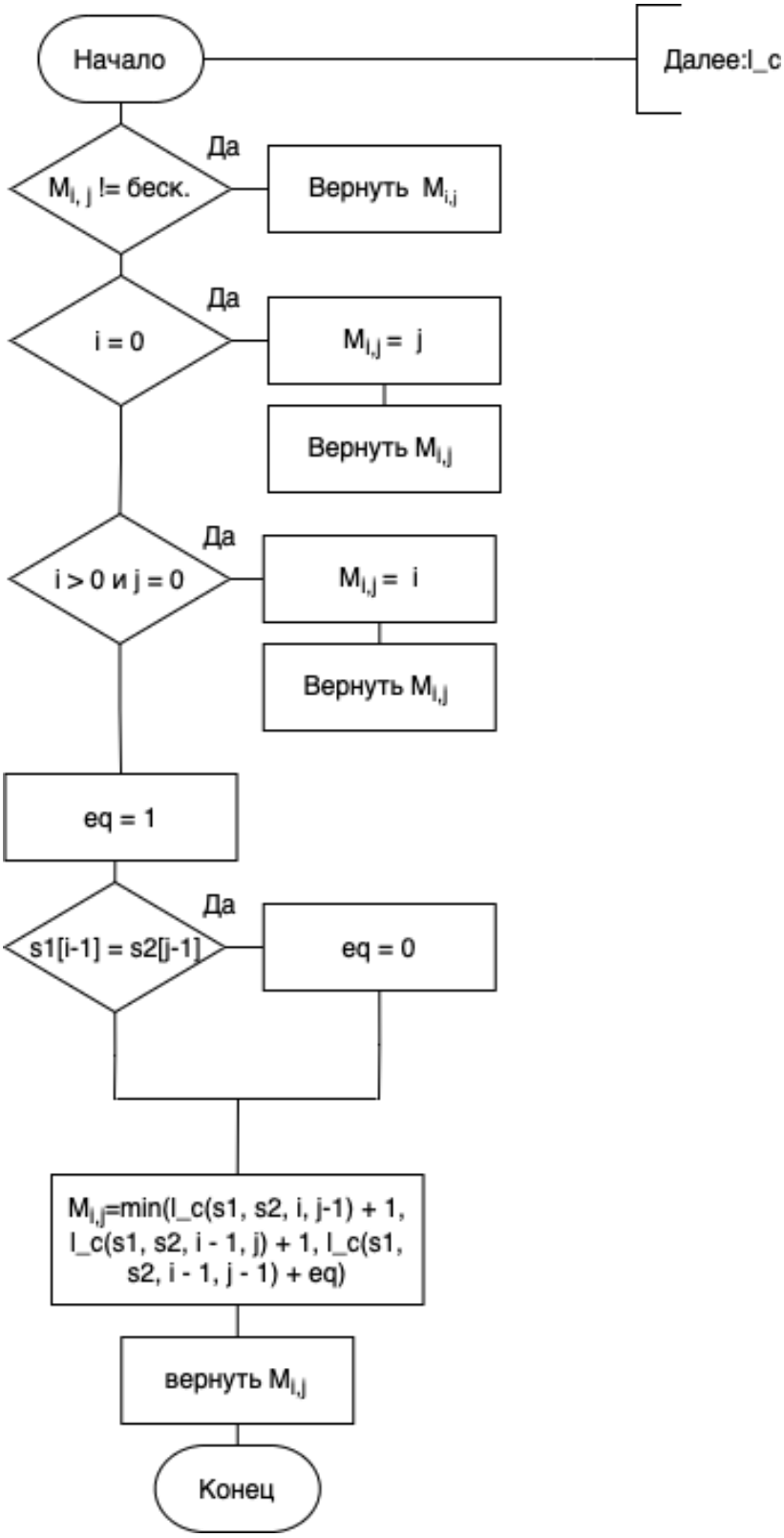

Рис. 2.4: Схема подпрограммы алгоритмы нахождения расстояния Левенштейна с кэшем

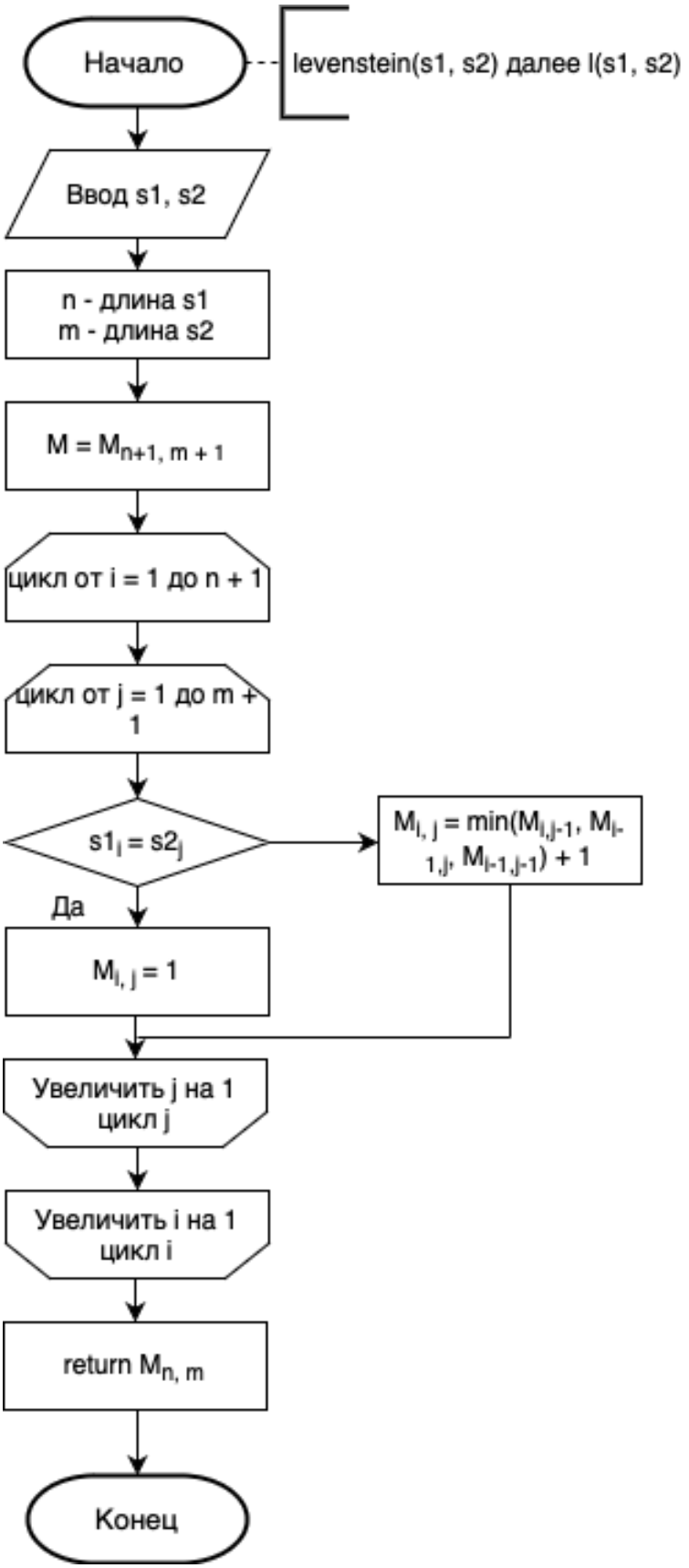

Рис. 2.5: Схема матричного алгоритма нахождения расстояния Левенштейна

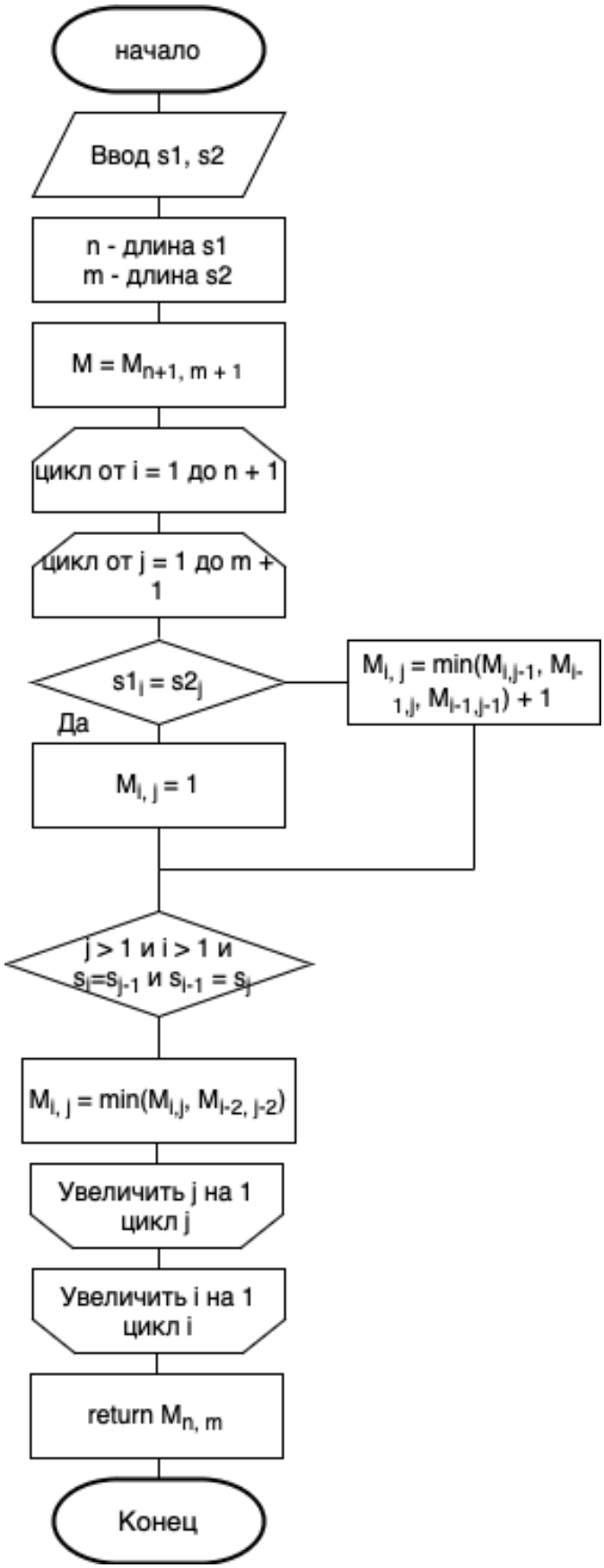

Рис. 2.6: Схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна

3 | Технологическая часть

В данном разделе приведены требования к программному обеспечению (далее – ПО), средства реализации и листинги кода

3.1 Требования к ПО

Требования к вводу

1. На вход подаются две строки
2. Заглавные и прописные буквы считаются разными

Требования к программе:

1. Две пустые строки - корректный ввод, программа не должна аварийно завершаться

3.2 Выбор языка программирования

Для реализации программ я выбрал язык программирования Rust [1], так как этот язык предоставляет как низкоуровневые интерфейсы, так и высокоуровневые. Также он является таким же быстрым, как и C++, но более безопасен. Среда разработки Visual Studio Code.

3.3 Листинги реализации алгоритма

В данных рекурсивных реализациях в связи с особенностями языка Rust, индексация строк идет от нуля и на каждый рекурсивный вызов в функцию передается подстрока от 1 элемента до n.

$$String = \{c_1, c_2, \dots c_n\}$$

$$Substring = \{c_2, \dots c_n\}, Substring \in String, n \in \mathbb{N}$$

Листинг 3.1: Функция нахождения расстояния Левенштейна рекурсивно

```
pub fn levenstein_rec(s1: &str, s2: &str) -> usize {
  let s1_len = s1.len();
  let s2_len = s2.len();

  if s1_len == 0 {
    return s2_len;
  }

  if s2_len == 0 {
    return s1_len;
  }

  if s1.chars().nth(0) == s2.chars().nth(0) {
    return levenstein_rec(&s1[1..], &s2[1..]);
  }

  let a = levenstein_rec(&s1[1..], &s2[1..]);
  let b = levenstein_rec(s1, &s2[1..]);
  let c = levenstein_rec(&s1[1..], &s2);

  return std::cmp::min(a, std::cmp::min(b, c)) + 1;
}
```

Листинг 3.2: Функция нахождения расстояния Дамерау-Левенштейна рекурсивно

```
pub fn damerau_levenstein_rec(s1: &str, s2: &str) -> usize {
  let s1_len = s1.len();
  let s2_len = s2.len();

  if s1_len == 0 {
    return s2_len;
  }

  if s2_len == 0 {
    return s1_len;
  }

  if s1.chars().nth(0) == s2.chars().nth(0) {
    return damerau_levenstein_rec(&s1[1..], &s2[1..]);
  }

  let a = damerau_levenstein_rec(&s1[1..], &s2[1..]);
  let b = damerau_levenstein_rec(s1, &s2[1..]);
  let c = damerau_levenstein_rec(&s1[1..], &s2);

  let mut d = usize::MAX;
  if s1_len > 1 && s2_len > 1 {
    d = damerau_levenstein_rec(&s1[2..], &s2[2..]);
  }

  return std::cmp::min(d, std::cmp::min(a, std::cmp::min(b, c))) + 1;
}
```

Листинг 3.3: Функция нахождения расстояния Левенштейна рекурсивно с кэшем

```rust
fn _levenstein_mem_rec(s1: &Vec<char>, s2: &Vec<char>, i: usize, j: usize, matrix: &mut Vec<Vec<usize>>) -> usize {
    if matrix[i][j] != usize::MAX {
        return matrix[i][j];
    }

    if i == 0 {
        matrix[i][j] = j;
        return matrix[i][j];
    }

    if j == 0 && i > 0 {
        matrix[i][j] = i;
        return matrix[i][j];
    }

    let mut eq = 1;

    if s1[i - 1] == s2[j - 1] {
        eq = 0;
    }

    matrix[i][j] = std::cmp::min(
        _levenstein_mem_rec(s1, s2, i, j - 1, matrix) + 1,
        std::cmp::min(
            _levenstein_mem_rec(s1, s2, i - 1, j, matrix) + 1,
            _levenstein_mem_rec(s1, s2, i - 1, j - 1, matrix) + eq
        )
    );

    return matrix[i][j];
}

pub fn levenstein_mem_rec(s1: &str, s2: &str) -> usize {
    let w1 = s1.chars().collect::<Vec<_>>();
    let w2 = s2.chars().collect::<Vec<_>>();

    let word1_length = w1.len();
    let word2_length = w2.len();

    let mut matrix = vec![vec![0; word1_length + 1]; word2_length + 1];

    for i in 0..word1_length + 1 {
        for j in 0..word2_length + 1 {
            matrix[i][j] = usize::MAX;
        }
    }

```

```
48      _levenstein_mem_rec(&w1, &w2, word1_length, word2_length, &mut matrix);
49
50      return matrix[word1_length][word2_length];
51  }
```

Листинг 3.4: Функция нахождения расстояния Левенштейна матрично

```rust
pub fn levenstein_iter(word1: &str, word2: &str) -> usize {
  // getting length of words
  let w1 = word1.chars().collect::<Vec<_>>();
  let w2 = word2.chars().collect::<Vec<_>>();

  let word1_length = w1.len() + 1;
  let word2_length = w2.len() + 1;

  let mut matrix = vec![vec![0; word1_length]; word2_length];

  for i in 1..word1_length {
    matrix[0][i] = i;
  }
  for j in 1..word2_length {
    matrix[j][0] = j;
  }

  for j in 1..word2_length {
    for i in 1..word1_length {
      let x: usize = if w1[i - 1] == w2[j - 1] {
        matrix[j - 1][i - 1]
      } else {
        1 + std::cmp::min(
          std::cmp::min(matrix[j][i - 1], matrix[j - 1][i]),
          matrix[j - 1][i - 1],
        )
      };
      matrix[j][i] = x;
    }
  }
  return matrix[word2_length - 1][word1_length - 1];
}
```

Листинг 3.5: Функция нахождения расстояния Дамерау-Левенштейна матрично

```rust
pub fn damerau_levenstein_iter(word1: &str, word2: &str) -> usize {
    // getting length of words
    let w1 = word1.chars().collect::<Vec<_>>();
    let w2 = word2.chars().collect::<Vec<_>>();

    let word1_length = w1.len() + 1;
    let word2_length = w2.len() + 1;

    let mut matrix = vec![vec![0; word1_length]; word2_length];

    for i in 1..word1_length {
        matrix[0][i] = i;
    }
    for j in 1..word2_length {
        matrix[j][0] = j;
    }

    for j in 1..word2_length {
        for i in 1..word1_length {
            let x: usize = if w1[i - 1] == w2[j - 1] {
                matrix[j - 1][i - 1]
            } else {
                1 + std::cmp::min(
                    std::cmp::min(matrix[j][i - 1], matrix[j - 1][i]),
                    matrix[j - 1][i - 1],
                )
            };

            matrix[j][i] = x;

            if (j > 1) && (i > 1) && (w1[i - 1] == w2[j - 2]) && (w1[i - 2]
               == w2[j - 1]) {
                matrix[j][i] = std::cmp::min(x, matrix[j - 2][i - 2] + 1);
            }
        }
    }
    return matrix[word2_length - 1][word1_length - 1];
}
```

В Таблице 3.1 приведены функциональные тесты для алгоритмов вычисления расстояния Левенштейна и Дамерау-Левенштейна.

Все тесты были пройдены успешно (ожидаемый результат совпал с фактическим).

Таблица 3.1: Функциональные тесты

Строка 1	Строка 2	Ожидаемый результат	
		Левенштейн	Дамерау-Левенштейн
Take	Took	3	3
Art	Atr	2	1
car	city	3	3
head	ehda	3	2
laptop	notebook	7	7
peek	peeks	1	1
rain	pain	1	1

4 | Исследовательская часть

В данном разделе будут представлены замеры времени работы реализаций алгоритмов и потребления памяти

4.1 Сравнительный анализ на основе замеров времени работы алгоритмов

Был проведен замер времени работы каждого из алгоритмов с помощью библиотеки Criterion [3]. Эта библиотека замеряет процессорное время выполнения функции и усредняет его. В таблице 4.1 содержится результат исследований.

Замер времени был выполнен со строками одинаковой длины. Длина строки – количество символов, содержащихся в этой строке.

Таблица 4.1: Время работы алгоритмов (в секундах)

Длина слов	Лев рек.	Лев итер.	Лев. рек. кэш	Дам-Лев рек.	Дам-Лев итер.
1	8,50E-08	2,48E-07	2,58E-07	3,20E-10	5,50E-07
2	1,08E-07	3,07E-07	3,63E-07	4,80E-08	4,93E-07
3	3,20E-07	4,26E-07	4,95E-07	2,77E-07	4,91E-07
4	1,30E-06	4,88E-07	6,46E-07	1,50E-06	5,00E-07
5	6,90E-06	7,41E-07	9,02E-07	8,40E-06	5,08E-07
6	3,65E-05	8,12E-07	1,00E-06	4,68E-05	5,31E-07
7	1,97E-04	9,86E-07	1,26E-06	2,64E-04	5,14E-07
8	1,00E-03	1,10E-06	1,47E-06	1,51E-03	5,20E-07
9	5,92E-03	1,22E-06	1,71E-06	8,72E-03	5,09E-07

На рис. 4.1 представлена зависимость времени работы реализаций алгоритмов от длины строк. Для удобства ось времени работы представлена в логарифмической шкале

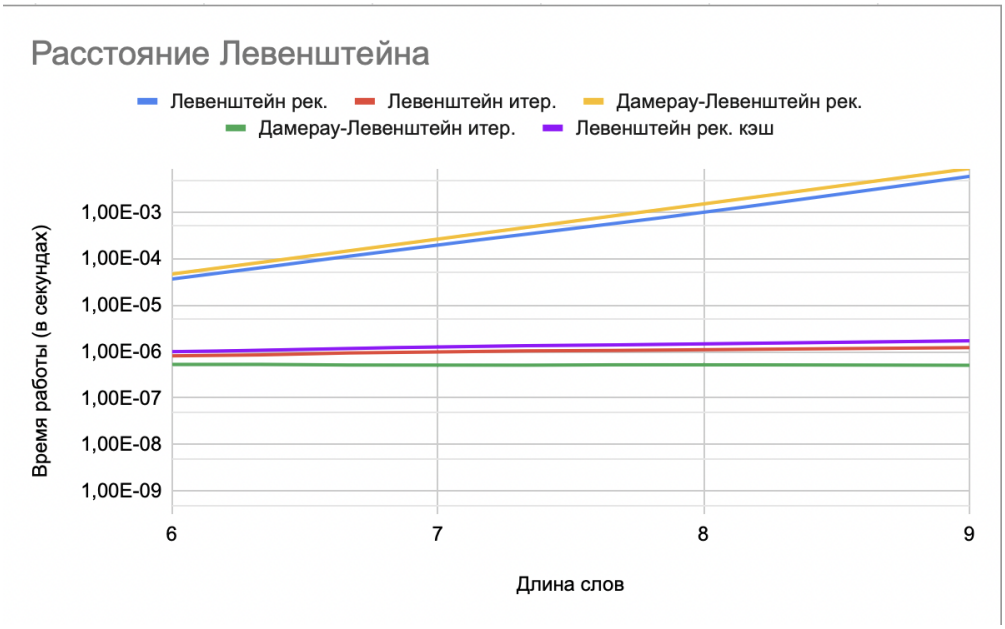

Рис. 4.1: Зависимость времени работы алгоритмов от длины строк

Наиболее эффективными по времени при маленькой длине слова являются рекурсивные реализации алгоритмов, но как только увеличивается длина слова, их эффективность резко снижается, что обусловлено большим количеством повторных рассчетов. Время работы алгоритма, использующего матрицу, намного меньше благодаря тому, что в нем требуется только (m + 1)*(n + 1) операций заполнения ячейки матрицы. Также установлено, что алгоритм Дамерау-Левенштейна работает немного дольше алгоритма Левенштейна, т.к. в нем добавлены дополнительные проверки, однако алгоритмы сравнимы по временной эффективности.

4.2 Использование памяти

Алгоритмы Левенштейна и Дамерау-Левенштейна не отличаются по использованию памяти, соответственно достаточно рассмотреть рекурсивный и матричный реализации этих алгоритмов.

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк, а на каждый вызов функции требуется еще 5 дополнительных переменных типа *usize*, соответственно, максимальный расход памяти

$$(Size(S_1) + Size(S_2) \cdot (2 \cdot Size(\text{string}) + 5 \cdot Size(\text{usize}))) \tag{4.1}$$

где Size - функция, возвращающая размер аргумента; string - строковый тип, usize - целочисленный, беззнаковый тип.

Использование памяти при итеративной реализации теоритически равно

$$(Size(S_1 + 1) \cdot Size(S_2 + 1)) \cdot Size(usize) + 2 \cdot Size(string) \tag{4.2}$$

Использование памяти рекурсивной реализации алгоритма Левенштейна с кэшем теоритически равно

$$(Size(S_1) + Size(S_2) \cdot (2 \cdot Size(\text{string}) + 5 \cdot Size(\text{usize})) + (Size(S_1 + 1) \cdot Size(S_2 + 1)) \cdot Size(usize)) \tag{4.3}$$

В данный момент отсутствуют инструменты для замера потребления памяти для языка Rust, поэтому подробные и конкретные замеры невозможны.

4.3 Вывод

Рекурсивный алгоритм Левенштейна работает на порядок дольше итеративных реализаций, время его работы увеличивается в геометрической прогрессии. На словах длиной 9 символов, матричная реализация превосходит рекурсивную в 4800 раз. Рекурсивные алгоритмы Левенштейна и Дамерау - Левенштейна сопостовимы по времени. Однако, использование кэша значительно ускоряет рекурсивный алгоритм, но он все еще не превосходит матричную реализацию.

Заключение

Был изучен метод динамического программирования на материале алгоритмов Левенштейна и Дамерау-Лев. Также изучены алгоритмы Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками, получены практические навыки раелизации указанных алгоритмов в матричной и рекурсивных версиях.

Экспериментально было подтверждено различие во временной эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработаного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк.

В результате исследований можно сделать вывод, что матричная реализация данных алгоритмов заметно выигрывает по времени при росте длины строк, следовательно более применима в реальных проектах.

При выполнение данной лабораторной работы были выполнены все цели и следующие задачи:

- Были изучены алгоритмов Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;
- Были применены методы динамического программирования для матричной реализации указанных алгоритмов;
- Были получены практические навыки реализации указанных алгоритмов: двух алгоритмов в матричной версии и одного из алгоритмов в рекурсивной версии;
- Были проведен сравнительный анализ линейной и рекурсивной реализаций выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);
- Было Экспериментально подтверждено различие во временной эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк;
- Были описаны и обоснованы полученные результаты в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

Литература

1. Блэнди Дж., Орендорф Дж. Программирование на языке Rust = Programming Rust. — ДМК Пресс, 2018. — 550 с. — ISBN 978-5-97060-236-2.

2. В. И. Левенштейн. Двоичные коды с исправлением выпадений, вставок и замещений символов. Доклады Академий Наук СССР, 1965. 163.4:845-848.

3. Criterion.rs - Statistics-driven benchmarking library for Rust [Электронный ресурс] `https://github.com/bheisler/criterion.rs` (дата обращения: 25 сентября 2021 г.)