МГТУ им. Баумана

Лабораторная работа №1

По курсу: "Анализ алгоритмов"

Расстояние Левенштейна

Работу выполнил: Саркисов Артём, ИУ7-53Б

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

Москва, 2020

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:

- исправления ошибок в слове(поисковая строка браузера)
- сравнения текстовых файлов утилитой diff
- в биоинформатике для сравнения генов, хромосом и белков

Целью данной лабораторной работы является изучение метода динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна.

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 двух 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

Задача по нахождению расстояния Левенштейна заключается в поиске минимального количества операций вставки/удаления/замены для превращения одной строки в другую.

При нахождении расстояния Дамерау — Левенштейна добавляется операция транспозиции (перестановки соседних символов).

Действия обозначаются так:

1. D (англ. delete) — удалить;
2. I (англ. insert) — вставить;
3. R (replace) — заменить;
4. M(match) - совпадение;

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 (1.1):

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ min(& \\ D(i,j-1) + 1, & \\ D(i-1,j) + 1, & j > 0, i > 0 \\ D(i-1,j-1) + m(S_1[i], S_2[j]) & \\ ), & \end{cases} \tag{1.1}$$

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

Расстояние Дамерау-Левенштейна вычисляется по следующей рекуррентной формуле (1.2):

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & i > 0, j = 0 \\ j, & i = 0, j > 0 \\ min \begin{cases} D(i,j-1)+1, \\ D(i-1,j)+1, \\ D(i-1,j-1)+m(S_1[i],S_2[i]), \\ D(i-2,j-2)+m(S_1[i],S_2[i]), \end{cases} & \text{, если } i,j > 0 \\ & \text{и } S_1[i] = S_2[j-1] \\ & \text{и } S_1[i-1] = S_2[j] \\ min \begin{cases} D(i,j-1)+1, \\ D(i-1,j)+1, \\ D(i-1,j-1)+m(S_1[i],S_2[i]), \end{cases} & \text{, иначе} \end{cases} \tag{1.2}$$

1.1 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

Требования к программе:

1. На вход подаются две строки
2. uppercase и lowercase буквы считаются разными
3. Две пустые строки - корректный ввод, программа не должна аварийно завершаться
4. Для всех алгоритмов выводиться процессорное время исполнения
5. Для всех алгоритмов кроме Левенштейна с рекурсивной реализацией должна выводиться матрица

2.1 Схемы алгоритмов

В данной части будут рассмотрены схемы алгоритмов.

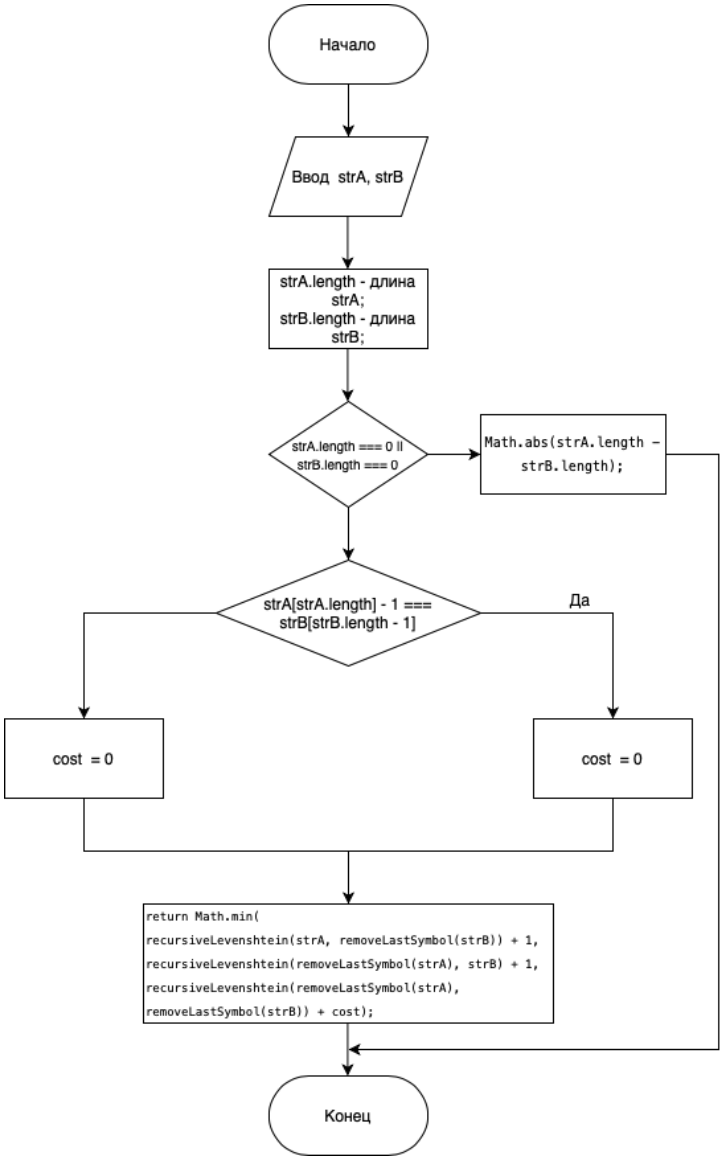


Рис. 2.1: Схема рекурсивного алгоритма нахождения расстояния Левенштейна

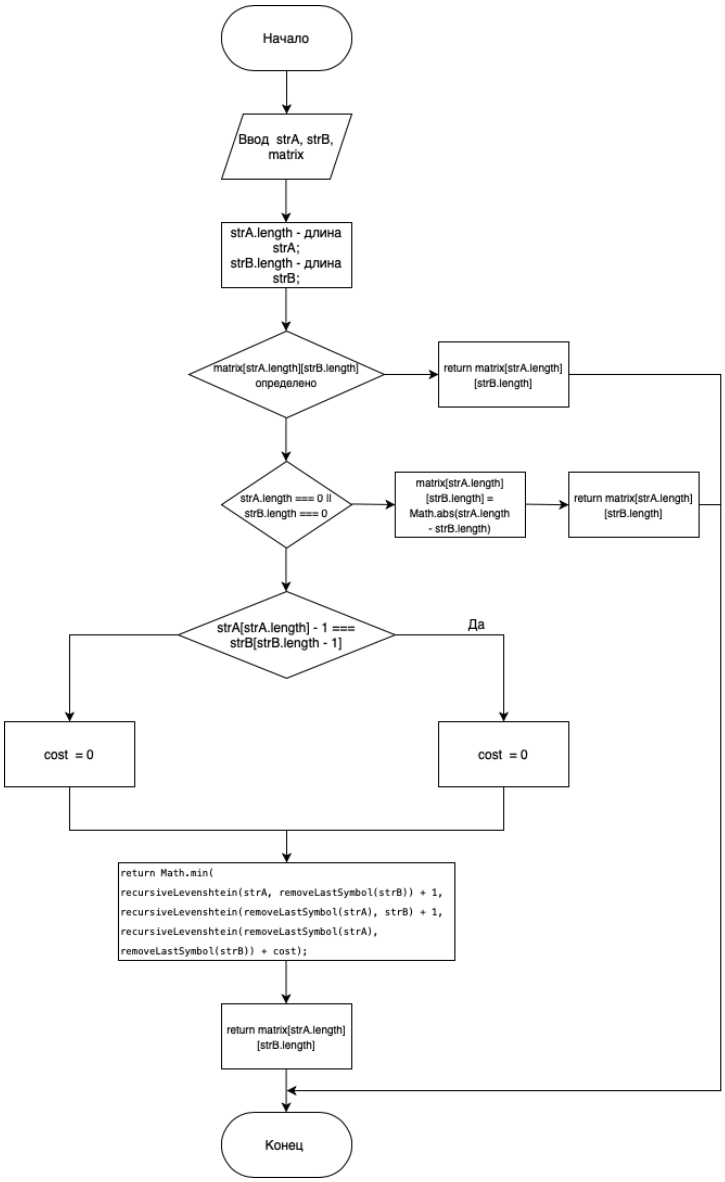


Рис. 2.2: Схема рекурсивного алгоритма нахождения расстояния Левенштейна с матричной оптимизацией

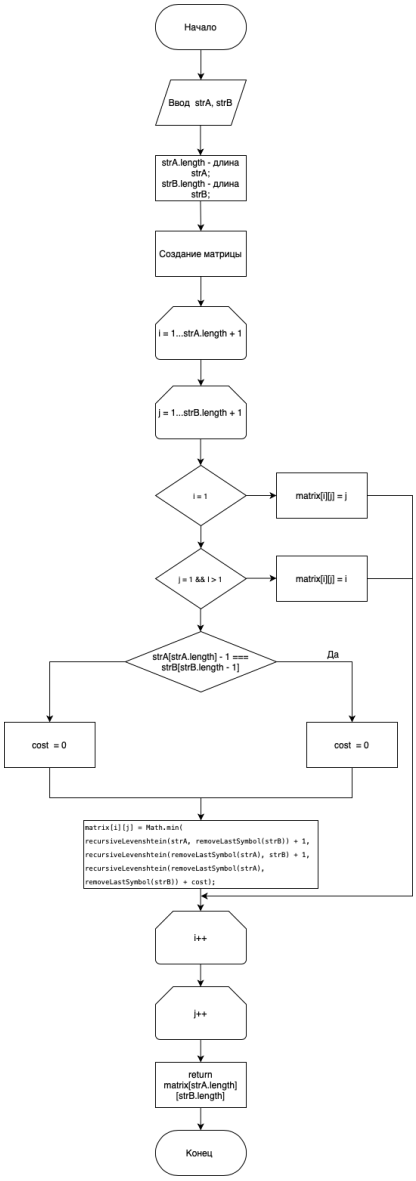


Рис. 2.3: Схема матричного алгоритма нахождения расстояния Левенштейна

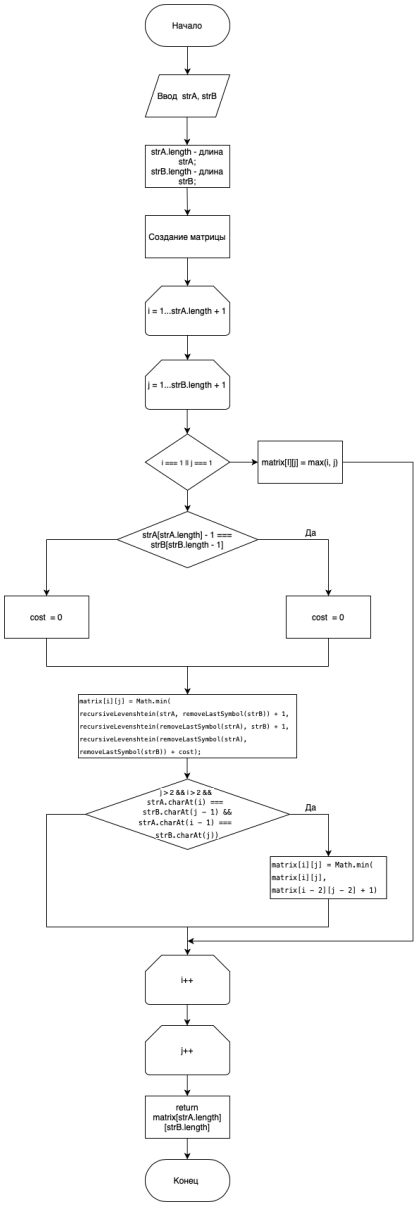


Рис. 2.4: Схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна

3 | Технологическая часть

3.1 Выбор ЯП

Для реализации программы был выбран язык программирования JavaScript в связи с потребностью практики разработки на нем. Среда разработки - VS Code.

3.2 Реализация алгоритма

Листинг 3.1: Функция нахождения расстояния Левенштейна рекурсивно

```
function recursiveLevenshtein(strA, strB) {
    if (strA.length === 0 || strB.length === 0) {
        return Math.abs(strA.length - strB.length);
    }
    let cost = (strA.charAt(strA.length - 1) === strB.
       charAt(strB.length - 1)) ? 0 : 1;
    return Math.min(recursiveLevenshtein(strA,
       removeLastSymbol(strB)) + 1,
                    recursiveLevenshtein(removeLastSymbol(
                       strA), strB) + 1,
                    recursiveLevenshtein(removeLastSymbol(
                       strA), removeLastSymbol(strB)) +
                       cost);
}
```

Листинг 3.2: Функция удаления последнего символа в строке

```
function removeLastSymbol(str) {
    return str.slice(0, -1);
```

```
}
```

Листинг 3.3: Функция нахождения расстояния Левенштейна рекурсивно с матрицей

```
function recursiveOptimizedLevenshtein(strA, strB, matrix)
    {
    if (typeof(matrix[strA.length][strB.length]) !== '
        undefined') {
        return matrix[strA.length][strB.length];
    }
    if (strA.length === 0 || strB.length === 0) {
        matrix[strA.length][strB.length] = Math.abs(strA.
            length - strB.length);
        return matrix[strA.length][strB.length];
    }
    let cost = (strA.charAt(strA.length - 1) === strB.
        charAt(strB.length - 1)) ? 0 : 1;
    matrix[strA.length][strB.length] = Math.min(
        recursiveOptimizedLevenshtein(strA,
            removeLastSymbol(strB), matrix) + 1,
        recursiveOptimizedLevenshtein(removeLastSymbol(strA
            ), strB, matrix) + 1,
        recursiveOptimizedLevenshtein(removeLastSymbol(strA
            ), removeLastSymbol(strB), matrix) + cost);
    return matrix[strA.length][strB.length];
}
```

Листинг 3.4: Функция нахождения расстояния Левенштейна матрично

```
function matrixLevenshtein(strA, strB, matrix) {
    for (let i = 0; i < strA.length + 1; i++) {
        matrix[i] = [];
        for (let j = 0; j < strB.length + 1; j++) {
            if (i === 0) {
                matrix[i][j] = j;
            } else if(j === 0 && i > 0) {
                matrix[i][j] = i;
            } else {
                let cost = (strA.charAt(i - 1) === strB.
                    charAt(j - 1)) ? 0 : 1;
```

```
11                  matrix[i][j] = Math.min(matrix[i][j - 1] +
                        1,
12                                          matrix[i - 1][j] +
                                                1,
13                                          matrix[i - 1][j -
                                                1] + cost);
14              }
15          }
16      }
17      return matrix[strA.length][strB.length];
18  }
```

Листинг 3.5: Функция нахождения расстояния Дамерау-Левенштейна матрично

```
1   function matrixDamerauLevenshtein(strA, strB, matrix) {
2       for (let i = 0; i < strA.length + 1; i++) {
3           matrix[i] = [];
4           for (let j = 0; j < strB.length + 1; j++) {
5               if (i === 0 || j === 0) {
6                   matrix[i][j] = Math.max(i, j);
7               } else {
8                   let cost = (strA.charAt(i - 1) === strB.
                        charAt(j - 1)) ? 0 : 1;
9                   matrix[i][j] = Math.min(matrix[i][j - 1] +
                        1,
10                      matrix[i - 1][j] + 1,
11                      matrix[i - 1][j - 1] + cost);
12                  if (j > 1 && i > 1 &&
13                      strA.charAt(i) === strB.charAt(j - 1)
                            &&
14                      strA.charAt(i - 1) === strB.charAt(j))
                            {
15                          matrix[i][j] = Math.min(matrix[i][j
                                ], matrix[i - 2][j - 2] + 1);
16                  }
17              }
18          }
19      }
20      return matrix[strA.length][strB.length];
21  }
```

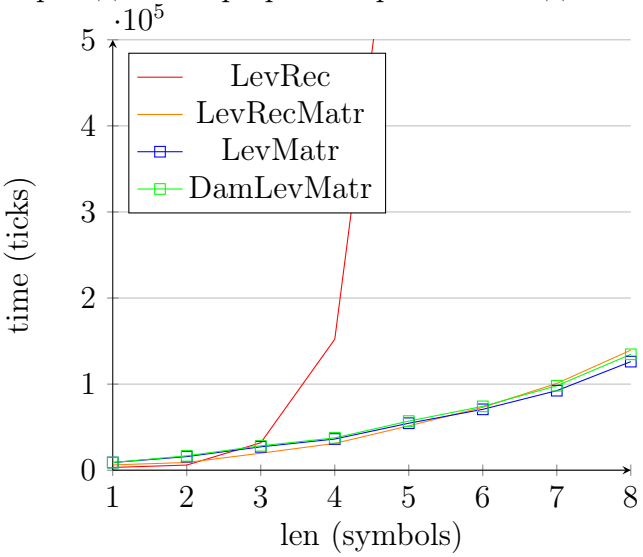
Листинг 3.6: Функция проверки транпозиции

```
1 func isTransposition(_ strA: String, _ strB: String, _ i:
    Int, _ j: Int) -> Bool {
2   return (strA[strA.index(strA.startIndex, offsetBy: i - 1)
      ] == strB[strB.index(strB.startIndex, offsetBy: j -
      2)] &&
3   strA[strA.index(strA.startIndex, offsetBy: i - 2)] ==
      strB[strB.index(strB.startIndex, offsetBy: j - 1)])
4 }
```

4 | Исследовательская часть

4.1 Сравнительный анализ на основе замеров времени работы алгоритмов

Был проведен замер времени работы каждого из алгоритмов.



Наиболее эффективными по времени при маленькой длине слова являются рекурсивные реализации алгоритмов. Показатели рекурсивного алгоритма Левенштейна при увеличении размера слов резко ухудшаются. Это связано с большим количеством повторов. Рекурсивная версия с матрицей не теряет своей производительности так как там исключаются повторы. Время работы алгоритма, использующего матрицу, намного меньше благодаря тому, что в нем требуется только (m + 1)*(n + 1) операций заполнения ячейки матрицы. Также видно, что алгоритм

Таблица 4.1: время исполнения в миллимекундах

len	Lev(R)	Lev(RM)	Lev(M)	DamLev(M)
1	0.025996	0.040448	0.022366	0.062549
2	0.048608	0.076069	0.033854	0.077381
3	0.106327	0.085223	0.028715	0.076817
4	0.502248	0.080871	0.026586	0.073056
5	0.862543	0.079230	0.030535	0.080112.
6	0.653954	0.048213	0.019119	0.046017
7	2.383116	0.074804	0.026301	0.066106
8	5.957557	0.064507	0.021543	0.002099
9	21.236497	0.084806	0.030479	0.072075
10	112.217936	0.088567	0.041352	0.088084

Таблица 4.2: Таблица тестовых данных

№	Слово №1	Слово №2	Ожидание	Результат
1			0 0 0 0	0 0 0 0
2	1234	2143	3 3 3 2	3 3 3 2
3	amm	add	2 2 2 2	2 2 2 2
4	error	dog	4 4 4 4	4 4 4 4
5		submission	10 10 10 10	10 10 10 10
6	mochombo	0	8 8 8 8	8 8 8 8
7	dfufdfd	fddfdffdd	5 5 5 4	5 5 5 4

Дамерау-Левенштейна работает немного дольше алгоритма Левенштейна, т.к. в нем добавлены дополнительные проверки, однако алгоритмы примерно одинаковы по временной эффективности.

4.2 Тесты

Проведем тестирование программы. В столбцах "Ожидание"и "Результат"4 числа соответсвуют рекурсивному алгоритму нахождения расстояния Левенштейна, рекурсивно-матричному алгоритму нахождения расстояния Левенштейна, матричному алгоритму расстояния Левенштейна, матричному алгоритму нахождения расстояния Дамерау-Левенштейна.

Заключение

Был изучен метод динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна. 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 делается вывод о том, что матричная реализация данных алгоритмов заметно выигрывает по времени при росте длины строк, следовательно более применима в реальных проектах.

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] JavaScript Programming Language [Электронный ресурс]. Режим доступа: https://https://developer.mozilla.org/ru/docs/Web/JavaScript/ (дата обращения: 20.09.2020).

[3] Absolute time. Режим доступа: https://https://developer.apple.com/documentation/kernel/1462446-machabsolutetime (дата обращения: 20.09.2020).

[4] MacOs Catalina [Электронный ресурс]. Режим доступа: https://https://www.apple.com/ru/macos/ catalina/ (дата обращения: 20.09.2020).

[5] Процессор Intel® Core™ i5 10 gen. [Электронный ресурс]. Режим доступа: https://https://www.intel.ru/content/www/ru/ru/products/docs/processors/core/ 10th-gen-processors.html (дата обращения: 20.09.2020).