МГТУ им. Баумана

Лабораторная работа №1

По курсу: "Анализ алгоритмов"

Расстояние Левенштейна

Работу выполнил: Гаврилов Дмитрий, ИУ7-56Б

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

Москва, 2019

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
- 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 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

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:

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

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

Расстояние Дамерау-Левенштейна вычисляется по следующей рекуррентной формуле:

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ min(& \\ D(i,j-1)+1, & \\ D(i-1,j)+1, & j > 0, i > 0 \\ D(i-1,j-1)+m(S_1[i],S_2[j]) & \\ D(i-2,j-2)+1, & \text{if } i,j > 1 \text{ and } a_i = b_{j-1}, a_{i-1} = b_j \\ ) & \end{cases}$$

2 | Конструкторская часть

Требования к вводу:

1. На вход подаются две строки
2. uppercase и lowercase буквы считаются разными

Требования к программе:

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

3 Технологическая часть

3.1 Выбор ЯП

В качестве языка программирования был выбран Java т.к. язык предоставляет обширный и удобный инструментарий для написания кода в Обьектно-ориентированном стиле и позволяет реализововать тестирование программы.

Время работы алгоритмов было замерено с помощью класса Instance из библиотеки java.time.

3.2 Сведения о модулях программы

Программа состоит из:

- DistanceBase.java - абстрактный класс, в который вынесена проверка на пустые строки, а так же базовый вызов методов для расчета растояния
- LowensteinDistance.java - класс с логикой для расчёта расстояния Левенштейна
- LowensteingDamerauDistance.java - класс с логикой для расчёта расстояния Дамерау-Левенштейна
- LowensteingRecursiveDistance.java - класс с логикой для расчёта расстояния Дамерау-Левенштейна рекурсивным способом
- Main.java - главный файл программы
- LowensteinDistanceTest.java - класс с тестами для расстояния Левенштейна

- LowensteingDamerauDistanceTest.java - класс с тестами для расстояния Дамерау-Левенштейна
- LowensteingRecursiveDistanceTest.java - класс с тестами для расстояния Дамерау-Левенштейна рекурсивным способом

Листинг 3.1: Базовый класс для вычисления расстояния

```
1  abstract class DistanceBase {
2      private int distance;
3      protected String firstWord;
4      protected String secondWord;
5
6      public DistanceBase(String firstWord, String
           secondWord) {
7          this.firstWord = firstWord;
8          this.secondWord = secondWord;
9      }
10
11
12     public int calculate() {
13         if (checkWorlds())
14             distance = calculateDistance();
15
16         return distance;
17     }
18
19     protected abstract int calculateDistance();
20
21     private boolean checkWorlds() {
22         if (firstWord.isEmpty() && secondWord.isEmpty())
               {
23             distance = 0;
24             return false;
25         }
26         else if (firstWord.isEmpty() || secondWord.
               isEmpty()) {
27             distance = 1;
28             return false;
29         }
30         return true;
```

```
31        }
32    }
```

Листинг 3.2: Функция нахождения расстояния Дамерау-Левенштейна рекурсивно

```
1  private int calculateRecursive(int firstWordLength, int
       secondWordLength) {
2          int cost;
3
4          if (firstWordLength == 0)
5              return secondWordLength;
6          if (secondWordLength == 0)
7              return firstWordLength;
8
9          if (firstWord.charAt(firstWordLength - 1) ==
               secondWord.charAt(secondWordLength - 1))
10             cost = 0;
11         else
12             cost = 1;
13
14
15         return Collections.min(Arrays.asList(
16                 calculateRecursive(firstWordLength - 1,
                       secondWordLength) + 1,
17                 calculateRecursive(firstWordLength,
                       secondWordLength - 1) + 1,
18                 calculateRecursive(firstWordLength - 1,
                       secondWordLength - 1) + cost
19         ));
20     }
```

Листинг 3.3: Класс нахождения расстояния Левенштейна матрично

```
1  @Override
2      protected int calculateDistance() {
3          matrix = new int[firstWord.length() + 1][secondWord
               .length() + 1];
4          fillMatrix();
5          findDistanceInMatrix();
6          return getResultFromMatrix();
```

```
7       }
8
9       protected void getNextElementInMatrix(int i, int j) {
10          int cost = calculateCost(i, j);
11
12          matrix[i][j] = Math.min(Math.min(matrix[i - 1][j] +
                1, matrix[i][j - 1] + 1), matrix[i - 1][j - 1]
               + cost);
13      }
14
15      protected int calculateCost(int i, int j) {
16          return (firstWord.charAt(i - 1) == (secondWord.
               charAt(j - 1))) ? 0 : 1;
17      }
18
19      private void findDistanceInMatrix() {
20          for (int i = 1; i <= firstWord.length(); i++) {
21              for (int j = 1; j <= secondWord.length(); j++)
                    {
22                  getNextElementInMatrix(i, j);
23              }
24          }
25      }
26
27      private void fillMatrix() {
28          fillMatrixFirstColumn();
29          fillMatrixFirstRow();
30      }
31
32      private void fillMatrixFirstColumn() {
33          for (int i = 0; i < firstWord.length(); i++) {
34              matrix[i][0] = i;
35          }
36      }
37
38      private void fillMatrixFirstRow() {
39          for (int i = 0; i < secondWord.length(); i++) {
40              matrix[0][i] = i;
41          }
42      }
```

```
43
44      private int getResultFromMatrix() {
45          return matrix[firstWord.length()][secondWord.length
                ()];
46      }
```

Листинг 3.4: Класс нахождения расстояния Дамерау-Левенштейна матрично

```
1  @Override
2      protected void getNextElementInMatrix(int i, int j) {
3          super.getNextElementInMatrix(i, j);
4
5          if (adjacentLetterEqual(i, j))
6              matrix[i][j] = Math.min(matrix[i][j], matrix[i
                   - 2][j - 2] + calculateCost(i, j));
7      }
8
9      private boolean adjacentLetterEqual(int i, int j) {
10         if (i > 1 && j > 1 &&
11                 firstWord.charAt(i - 1) == secondWord.
                       charAt(j - 2) &&
12                 firstWord.charAt(i - 2) == secondWord.
                       charAt(j - 1)) {
13             return true;
14         }
15         return false;
16     }
```

3.3 Тесты

Тестирование было организовано с помощью библиотеки **JUnit**. Было создано слудеющие вариации тестов:

Сравнивались результаты функции с реальным результатом для разных типов строк:

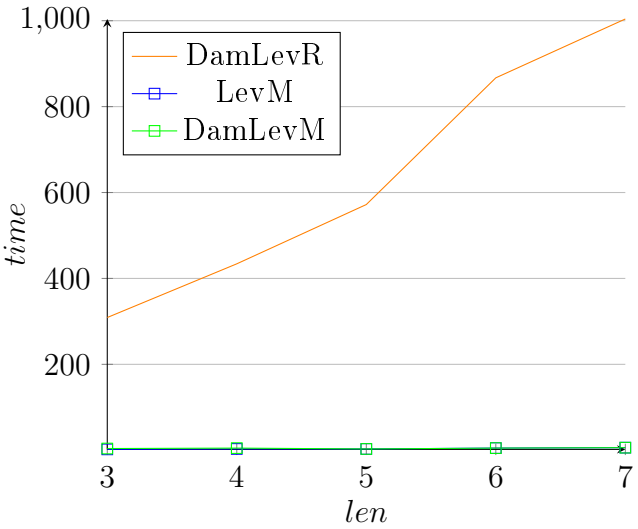
- - Строк с одним различием
- - Строк с множеством различиев

- - Одинаковых строк
- - Пустых строк

4 Исследовательская часть

Был проведен замер времени работы каждого из алгоритмов.

len	Lev(M)	DamLev(M)	DamLev(R)
3	2	4	309
4	3	5	434
5	3	3	572
6	5	5	867
7	6	7	1004



Рекурсивная реализация занимает несравнимо больше времени чем матричная. При увеличении длины строк выигрышность по времени матричного варианта становится в 1,000 раз быстрее уже при длине в 7 символов.

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 реализации 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 пришел к выводу, что матричная реализация данных алгоритмов заметно выигрывает по времени при росте длины строк.