Kromp Florian, Plangger Richard; 12.12.2013

Exercise 5

For the solution of Exercise 5, we chose to implement the Jaro as well as the Jaro-Winkler metric.

The source code is given in the file jaro.pl, we implemented the algorithm in the scripting language perl.

As a sample input, we chose two strings: <u>larmen</u> and <u>lernen</u>

Now we explain the algorithm based on the two sample-strings:

First, we calculate the Jaro-Distance. For this purpose, the characters, that are common in both strings with respect to a given maximum distance, are computed using the function getCommonCharacters. As the maximum distance between a character in the first string and the corresponding character in the second string, we defined (based on the implementation of SimMetric) the half of length of the shorter string + 1, rounded to the next lower integer.

```
my $halflen = int((min(length($string1), length($string2)) / 2) + 1);
```

So in this case, the max distance is 6/2 + 1 = 4. So every characters of the second string, in the range of -4 until +3 (abortion condition), are proven in string 2, and if one being equal is found, the character is added to the output array and replaced in the second string by a character ‚#'.

So, for this two strings, the procedure looks as follows:

character of the first string --> if found in the second string insiderange of -4/+3 positions -->the character is added to the outputarray @characters and replaced by ‚#'.

I describe the states after performing the steps when looping through the characters of string1:

String1: lernen, String2: larmen

Range is 6/2 +1 = 4 -> -4 / +3

State at start: @character = (); @string1 = („l“,“e“,“r“,“n“,“e“,“n“); @copy = („l“,“a“,“r“,“m“,“e“,“n“);

State	Position - @string1	Range of chars - @copy	Match?	@characters	@copy neu
1. Iteration	(„l“,“e“,“r“,“n“,“e“,“n“)	(„l“,“a“,“r“,“m“,“e“,“n“)	yes	(„l“)	(„#“,“a“,“r“,“m“,“e“,“n“)
2.Iteration	(„l“,“e“,“r“,“n“,“e“,“n“)	(„#“,“a“,“r“,“m“,“e“,“n“)	yes	(„l“,“e“)	(„#“,“a“,“r“,“m“,“#“,“n“)
3. Iteration	(„l“,“e“,“r“,“n“,“e“,“n“)	(„#“,“a“,“r“,“m“,“#“,“n“)	yes	(„l“,“e“,“r“)	(„#“,“a“,“#“,“m“,“#“,“n“)
4. Iteration	(„l“,“e“,“r“,“n“,“e“,“n“)	(„#“,“a“,“#“,“m“,“#“,“n“)	yes	(„l“,“e“,“r“,“n“)	(„#“,“a“,“#“,“m“,“#“,“#“)
5. Iteration	(„l“,“e“,“r“,“n“,“e“,“n“)	(„#“,“a“,“#“,“m“,“#“,“#“)	no	(„l“,“e“,“r“,“n“)	(„#“,“a“,“r#“,“m“,“#“,“#“)
6. Iteration	(„l“,“e“,“r“,“n“,“e“,“n“)	(„#“,“a“,“#“,“m“,“#“,“#“)	no	(„l“,“r“,“n“,“e“)	(„#“,“a“,“#“,“m“,“#“,“#“)

Now we call the function again, but change the strings:

String1: larmen, String2: lernen

Range is 6/2 +1 = 4

State at start: @character = (); @string1 = („l“,“a“,“r“,“m“,“e“,“n“); @copy = („l“,“e“,“r“,“n“,“e“,“n“);

State	Position - @string1	Range of chars - @copy	Match?	@characters	@copy neu
1. Iteration	(„l“,“a“,“r“,“m“,“e“,“n“)	(„l“,“e“,“r“,“n“,“e“,“n“)	yes	(„l“)	(„#“,“e“,“r“,“n“,“e“,“n“)
2.Iteration	(„l“,“a“,“r“,“m“,“e“,“n“)	(„#“,“e“,“r“,“n“,“e“,“n“)	no	(„l“)	(„#“,“e“,“r“,“n“,“e“,“n“)
3. Iteration	(„l“,“a“,“r“,“m“,“e“,“n“)	(„#“,“e“,“r“,“n“,“e“,“n“)	yes	(„l“,“r“)	(„#“,“e“,“#“,“n“,“e“,“n“)
4. Iteration	(„l“,“a“,“r“,“m“,“e“,“n“)	(„#“,“e“,“#“,“n“,“e“,“n“)	no	(„l“,“r“)	(„#“,“e“,“#“,“n“,“e“,“n“)
5. Iteration	(„l“,“a“,“r“,“m“,“e“,“n“)	(„#“,“e“,“#“,“n“,“e“,“n“)	yes	(„l“,“r“,“e“)	(„#“,“e“,“#“,“n“,“#“,“n“)
6. Iteration	(„l“,“a“,“r“,“m“,“e“,“n“)	(„#“,“e“,“#“,“n“,“#“,“n“)	yes	(„l“,“r“,“e““n“)	(„#“,“e“,“#“,“n“,“#“,“#“)

So the amount of common characters is 4 in both cases.

The algorithm now computes the amount of characters, that stay the same at the same position (=the amount of transpositions) for the common characters: this is the number 2 („l“,“r“).

We found an implementation, where the amount of transpositions is divided by two, but this doesn’t deliver a value of the Jaro-distance between zero and one; so we decided, to take the approach of an other paper, found on wikipedia, where the amount of transpositions is not divided by 2.

Then we calculate the Jaro-Distance:

($laengeCommon1 / length($string1)) + ($laengeCommon2 / length($string2)) + (($laengeCommon1 - $transpositions) / ($laengeCommon1)) / 3

This is (4/6 + 4/6 + (4-2)/4) / 3 = (2/3 + 2/3 + 1/2) / 3 = 0.61111

Now we can compute the Jaro-Winkler Metric, which is an extension of the Jaro-Winkler distance:

$jaro + ($prefix_len * $prefix_scale * (1 - $jaro))

The algorithm counts the amount of matching characters at the beginning of both strings, until the first mismatching character is found. This is weighted by the term prefix_scale. So when computing the jaro-Winkler distance, you can weight the first matching characters.

For our example, we choose a scale of 0.2, perfix_len is 1 („l“).

so the Jaro-Winkler Metric is 0.61111 + (1 * 0.2 * (1-0.6111) = 0.68888

If we choose a higher scale (for example 0.4), we get a value of 0.76666

If we had more matching characters at the beginning, we would achieve an proportionally higher value for the Jaro-Winkler-Metric.

Here is an example, where the Jaro distance returns zero (and so the Jaro-Winkler metric):

String1: moddy, String2: tmobiley

Range is 5/2+1 = 3.5 -> 3 Range: -3 / +2

State at start: @character = (); @string1 = („m“,“o“,“o“,“d“,“y“); @copy = („t“,“m“,“o“,“b“,“i“,“l“,“e“,“y“);

State	Position - @string1	Range of chars - @copy	Match?	@characters	@copy neu
1. Iteration	(„m“,“o“,“o“,“d“,“y“)	(„t“,“m“,“o“,“b“,“i“,“l“,“e“,“y“)	yes	(„m“)	(„t“,“#“,“o“,“b“,“i“,“l“,“e“, “y“)
2.Iteration	(„m“,“o“,“o“,“d“,“y“)	(„t“,“#“,“o“,“b“,“i“,“l“,“e“,“y“)	yes	(„m“,“o“)	(„t“,“#“,“#“,“b“,“i“,“l“,“e“, “y“)
3. Iteration	(„m“,“o“,“o“,“d“,“y“)	(„t“,“#“,“#“,“b“,“i“,“l“,“e“, “y“)	no	(„m“,“o“)	(„t“,“#“,“#“,“b“,“i“,“l“,“e“, “y“)
4. Iteration	(„m“,“o“,“o“,“d“,“y“)	(„t“,“#“,“#“,“b“,“i“,“l“,“e“, “y“)	no	(„m“,“o“)	(„t“,“#“,“#“,“b“,“i“,“l“,“e“, “y“)
5. Iteration	(„m“,“o“,“o“,“d“,“y“)	(„t“,“#“,“#“,“b“,“i“,“l“,“e“, “y“)	no	(„m“,“o“)	(„t“,“#“,“#“,“b“,“i“,“l“,“e“, “y“)

No we call the function again, but change the order of the strings:

State at start: @character = (); @string1 =(„t“,“m“,“o“,“b“,“i“,“l“,“e“,“y“); @copy = („m“,“o“,“o“,“d“,“y“);

State	Position - @string1	Range of chars - @copy	Match?	@characters	@copy neu
1. Iteration	(„t“,“m“,“o“,“b“,“i“,“l“,“e“,“y“)	(„m“,“o“,“o“,“d“,“y“)	no	()	(„m“,“o“,“o“,“d“,“y“)
2.Iteration	(„t“,“m“,“o“,“b“,“i“,“l“,“e“,“y“)	(„m“,“o“,“o“,“d“,“y“)	yes	(„m“)	(„#“,“o“,“o“,“d“,“y“)
3. Iteration	(„t“,“m“,“o“,“b“,“i“,“l“,“e“,“y“)	(„#“,“o“,“o“,“d“,“y“)	yes	(„m“,“o“)	(„#“,“#“,“o“,“d“,“y“)
4. Iteration	(„t“,“m“,“o“,“b“,“i“,“l“,“e“,“y“)	(„#“,“#“,“o“,“d“,“y“)	no	(„m“,“o“)	(„#“,“#“,“o“,“d“,“y“)
5. Iteration	(„t“,“m“,“o“,“b“,“i“,“l“,“e“,“y“)	(„#“,“#“,“o“,“d“,“y“)	no	(„m“,“o“)	(„#“,“#“,“o“,“d“,“y“)
6. Iteration	(„t“,“m“,“o“,“b“,“i“,“l“,“e“,“y“)	(„#“,“#“,“o“,“d“,“y“)	no	(„m“,“o“)	(„#“,“#“,“o“,“d“,“y“)
7. Iteration	(„t“,“m“,“o“,“b“,“i“,“l“,“e“,“y“)	(„#“,“#“,“o“,“d“,“y“)	no	(„m“,“o“)	(„#“,“#“,“o“,“d“,“y“)
8. Iteration	(„t“,“m“,“o“,“b“,“i“,“l“,“e“,“y“)	(„#“,“#“,“o“,“d“,“y“)	yes	(„m“,“o“,“y“)	(„#“,“#“,“o“,“d“,“#“)

The length of the array common1String is 2, the length of the common2String is 3, so the algorithm returns 0.0.