**Algorithm Study Template**

**Algorithm**: Damerau-Levenshtein Distance

**aka**: Optimal String Alignment Distance

**Techniques**: String Metric/String Distance

**Categories**: Approximate String Matching, Bioinformatics

**Problem**: The original motivation behind the algorithm was to measure distance between human misspellings to improve applications like spell checkers, but it has also been used in different areas, such as for measuring the variation between DNA sequences in the field of bioinformatics.

**Applications**: Three of the most widely used applications are natural language processing, measuring variations in DNA sequences, and detecting fraud. Specifically, when detecting fraud, the algorithm may check manually entered data for errors intentionally made by a fraudster. For example, when entering the name of a vendor to be paid for a service, a nefarious employee may enter “Rich Hier State Services” instead of “Rich Heir Estate Services” because they look almost the same at a glance. Then the employee could funnel money to a phony bank account in the name of the falsified vendor. The algorithm would catch the variation in the spelling of the vendor name (two letters transposed and one letter dropped) and alert appropriate personnel of the fraudulent activity.

**References**:

* <http://en.wikipedia.org/wiki/String_metrics>
* <http://en.wikipedia.org/wiki/Approximate_string_matching>
* <http://profs.sci.univr.it/~liptak/ALBioinfo/files/levenshtein66.pdf>
* <http://fuzzy-string.com/Compare/>
* <http://en.wikibooks.org/wiki/Algorithm_Implementation/Strings/Levenshtein_distance>
* <http://scarcitycomputing.blogspot.com/2013/04/damerau-levenshtein-edit-distance.html>

**Implementation details**:

* **Big Idea**: This algorithm finds the “distance” between two strings by counting the minimum number of operations needed to transform one string into another, where an operation is defined as an insertion, deletion, or substitution of a single character, or a transposition of two adjacent characters.
* **Description**:

The algorithm starts by creating a two-dimensional array as a table to hold integer values representing what operations are needed at each index of both strings to transform the first string into the second. The table first has its left-most column filled by the index numbers of the first string. Then, the top row is filled by the index numbers of the second string.

Now the rest of the table is iterated through until each cell displays the cumulative number of changes required to arrive there, as well as what type of change is required to arrive there. There will be a diagonal change path moving from the top left corner of the table to the bottom right corner. This path shows the number of changes that have been found, and the value in the bottom-right corner represents the total number of changes.

The number of changes is affected by the need to perform the operations insert, delete, substitute, and transpose on characters from the first string to create the second string. The change path can be difficult to understand, but there is a method to the madness. When comparing two characters in the table, if an insertion is needed, we move one column to the right and add the letter from the second string. If a deletion is needed, we move down one row and remove the letter from the first string. If a substitution is needed, we move down diagonally to the right and replace the letter from the first string with the letter from the second string. If a transposition is needed, we move down diagonally to the right twice, counting only one change, and swapping the letter from the first string with the next letter from that same string. The number of changes starts as zero, but if the two characters being compared on the change path are found to be anything but equal, the number in that spot on the change path is incremented by one.

That covers the change path, but how does the rest of the table get filled? In short, the cells that aren’t on the change path have their values determined by the change path. The easiest way to think of it is that the cells above the change path in a given column increase by a value of one each cell until the top of the table is reached. Similarly, the cells below the change path in a given column increase by a value of one each cell until the bottom of the table is reached. This means that the smallest values in the table occur in the change path. The two corners of the table which are not included in the change path (bottom-left and top-right) are where the largest values in the table can be seen.

Still confused? I don’t blame you. A written explanation of the table doesn’t fully do it justice. I recommend visiting <http://fuzzy-string.com/Compare/Transform.aspx?r=A+CAT&q=A+ABCT> for a nicely drawn and explained example of a distance table made by the Damerau-Levenshtein Distance algorithm. In this specific example “A CAT” is being compared to “A ABCT”. The change path in this example consists of two diagonals because the strings have different lengths. The explanation on that site is largely what the rest of my description of this algorithm is based on.

* **Pseudo-code**: (adapted from <http://en.wikipedia.org/wiki/Damerau%E2%80%93Levenshtein_distance> )

**int** damerauLevenshtein(**char** str1[1..lenStr1], **char** str2[1..lenStr2])

*// d is a table with lenStr1+1 rows and lenStr2+1 columns*

**declare** **int** d[0..lenStr1, 0..lenStr2]

*// i and j are used to iterate over str1 and str2*

**declare** **int** i, j, cost

*//for loop is inclusive, need table 1 row/column larger than string length.*

**for** i **from** 0 **to** lenStr1

d[i, 0] := i

**for** j **from** 1 **to** lenStr2

d[0, j] := j

*//Pseudo-code assumes string indices start at 1, not 0.*

*//When implementing, make sure to start comparing at 1st letter of strings.*

**for** i **from** 1 **to** lenStr1

**for** j **from** 1 **to** lenStr2

**if** str1[i] = str2[j] **then** cost := 0

**else** cost := 1

d[i, j] := minimum(

d[i-1, j ] + 1, *// deletion*

d[i , j-1] + 1, *// insertion*

d[i-1, j-1] + cost *// substitution*

)

**if**(i > 1 **and** j > 1 **and** str1[i] = str2[j-1] **and** str1[i-1] = str2[j]) **then**

d[i, j] := minimum(

d[i, j],

//transposition

d[i-2, j-2] + cost

)

**return** d[lenStr1, lenStr2]

* **Specific implementation**: (See DamerauLevenshtein.java, which is based on the above pseudo-code, except that I used indices starting at zero instead of one.)

**Correctness**:

**Theoretical**: This version of the Damerau-Levenshtein distance algorithm is known as the optimal string alignment algorithm because it does not edit any substring more than once. This means that adjacent transpositions are not handled, and is why this version of the algorithm is sometimes called the restricted edit distance. It is a direct extension of the Levenshtein distance algorithm that has an additional condition to check for transpositions. If the characters are equal, the algorithm continues to move through the table. If an insertion, deletion, substitution, or transposition is needed, this is annotated in the table and the number of changes is incremented in the change path. The final count in the change path is taken to be the total number of changes.

**Empirical**: The program I wrote allows the user to perform two different tests. The first test allows the user to input two non-numeric strings and compute both the Damerau-Levenshtein distance and the Levenshtein distance, along with execution times. I included the Levenshtein distance for comparison purposes but chose not to discuss it as a separate algorithm because it is merely the Damerau-Levenshtein algorithm with one less step (for transposition). Of course, I included checks to ensure that the user must input non-numeric strings, with the penalty for unacceptable input being an error message and program termination. I chose to allow punctuation marks because they are handled well by the algorithm and may be desired by someone using the program.

The second test the user can run is a mock DNA sequence analysis. The user specifies how large they want the compared sequences to be, and the program generates two sequences of that size by randomly inserting ‘A’, ‘C’, ‘G’, and ‘T’ into a string until the desired sizes are reached. Then the two string metric algorithms are run and their results and execution times are reported. Again, I included checks to ensure that unacceptable input cannot break the program, with error messages and program termination as the penalty for bad input.

There is something I must note about the DNA analysis test. I had originally planned to compare sequences that were millions or even billions of characters long, but I quickly discovered that this was not possible on my computer with this version of the algorithm. The issue is that I run out of heap space when the two-dimensional integer array is created for the distance table. I ran some numbers and found out that an integer array with dimensions 100,000x100,000 would require over twice the amount of memory in my computer. Because of this, I could only test with numbers in the ten thousands, even when bumping my heap size up to six gigabytes. With that in mind, the sequence sizes below are the largest that I could get to work.

Test 1:

Size: 38000

Damerau-Levenshtein Distance: 19422

Levenshtein Distance: 19663

Test 2:

Size: 37750

Damerau-Levenshtein Distance: 19284

Levenshtein Distance: 19556

Test 3:

Size: 37500

Damerau-Levenshtein Distance: 19118

Levenshtein Distance: 19381

Test 4:

Size: 37000

Damerau-Levenshtein Distance: 18897

Levenshtein Distance: 19144

Although the test data isn’t as large as I would have liked, there are still conclusions that can be drawn from the results. The Damerau-Levenshtein distance was always smaller than the Levenshtein distance. This has to be because of transpositions that occurred with Damerau-Levenshtein, but weren’t supported by Levenshtein. Even with reasonably large sequences, there are only four characters in use, so the possibility of characters within one sequence only needing to be swapped (or transposed) to match the other sequence is clearly present. Both distances were always fairly large, so there must not have been too many transpositions being found, but there were enough to make a visible difference between the distances.

I also did some testing with the option to input my own strings to compare:

Test 1:

Your first string: fish

Your second string: fihs

Damerau-Levenshtein Distance: 1

Levenshtein Distance: 2

Test 2:

Your first string: bunny

Your second string: funni

Damerau-Levenshtein Distance: 2

Levenshtein Distance: 2

Test 3:

Your first string: fast taco

Your second string: fsat atco

Damerau-Levenshtein Distance: 2

Levenshtein Distance: 4

Here, we see that the Damerau-Levenshtein distance isn’t always less than the Levenshtein distance. In test two, we see that two substitutions are required, whether transpositions are allowed or not, resulting in the two distances being equal. In tests one and three we see situations where transpositions are present. The Damerau-Levenshtein algorithm detects this and finds a way to make the two strings match in fewer operations than the Levenshtein algorithm, which is more likely to take the approach of using four substitutions in test three than two transpositions, for example.

**Performance**:

**Theoretical**: The complexity is approximately O(M\*N).

**Empirical**: The following execution times are only measuring the execution times of the Damerau-Levenshtein and Levenshtein distance algorithms. No printed output or other “noise” is intentionally measured.

Test 1:

Size: 38000

Damerau-Levenshtein Distance computed in: 28886.604 milliseconds.

Levenshtein Distance computed in: 25512.688 milliseconds.

Test 2:

Size: 37750

Damerau-Levenshtein Distance computed in: 28177.09 milliseconds.

Levenshtein Distance computed in: 44713.669 milliseconds.

Test 3:

Size: 37500

Damerau-Levenshtein Distance computed in: 29439.15 milliseconds.

Levenshtein Distance computed in: 21259.494 milliseconds.

Test 4:

Size: 37000

Damerau-Levenshtein Distance computed in: 28239.363 milliseconds.

Levenshtein Distance computed in: 15199.223 milliseconds.

In these DNA sequence tests, we see that the Damerau-Levenshtein algorithm almost always has a longer execution time than the Levenshtein algorithm. This is most likely because of the extra condition involved in checking for and handling transpositions. There was one case where the Levenshtein algorithm took much longer to execute, but this could have been thrown off by other applications running on my computer at the same time. Otherwise, I suppose it could have been because of a drastic number of operations to be performed that weren’t transpositions, due to the randomness of the input sequences.

Here are the execution times from my manually entered string tests:

Test 1:

Damerau-Levenshtein Distance computed in: 0.42 milliseconds.

Levenshtein Distance computed in: 0.026 milliseconds.

Test 2:

Damerau-Levenshtein Distance computed in: 0.427 milliseconds.

Levenshtein Distance computed in: 0.04 milliseconds.

Test 3:

Damerau-Levenshtein Distance computed in: 0.291 milliseconds.

Levenshtein Distance computed in: 0.109 milliseconds.

In these cases, the Damerau-Levenshtein distance always takes more time to compute. I believe this is because of the extra logic involved in checking for and performing transpositions. In smaller test cases like these, one extra step to perform almost guarantees longer execution time. In test two, we see that the times are almost equal, most likely because that was the test where only two substitutions were needed and no transpositions were present. Because of that condition, both algorithms executed almost identically in that case.

**Anecdotes**: <none>

**History**: Originally, Vladimir Levenshtein wrote the Levenshtein Distance algorithm to find the edit distance between two strings using three operations: insertion, deletion, and substitution. Sometime later, the work of Frederick Damerau was applied to the algorithm and the fourth operation, transposition, was added. Thus, the name Damerau-Levenshtein Distance refers to Levenshtein’s edit distance, with an added condition to allow for Damerau’s transpositions.

**Variations**: Damerau-Levenshtein Distance is itself a variation of Levenshtein Distance. Beyond that, I cannot identify any variations of Damerau-Levenshtein distance.

**Alternatives**: In the field of bioinformatics, the Needleman-Wunsch algorithm and the Smith-Waterman algorithm are both closely related to the Damerau-Levenshtein Distance algorithm in their ability to compare and align sequences.

**Credits:**

* <http://en.wikipedia.org/wiki/Levenshtein_distance>
* <http://en.wikipedia.org/wiki/Damerau%E2%80%93Levenshtein_distance>
* <http://stackoverflow.com/questions/5683327/how-to-generate-a-random-string-of-20-characters>
* <http://fuzzy-string.com/Compare/Transform.aspx?r=A+CAT&q=A+ABCT>