Damerau–Levenshtein distance

From Wikipedia, the free encyclopedia

WIKIPEDIA
The Free Encyclopedia
Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store
Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page
Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page
Print/export
Create a book
Download as PDF
Printable version
Languages
Español
Français
Polski
Русский
Српски / srpski
ไทย
Edit links

In information theory and computer science, the **Damerau–Levenshtein distance** (named after Frederick J. Damerau and Vladimir I. Levenshtein[1][2][3]) is a distance (string metric) between two strings, i.e., finite sequence of symbols, given by counting the minimum number of operations needed to transform one string into the other, where an operation is defined as an insertion, deletion, or substitution of a single character, or a transposition of two **adjacent** characters. In his seminal paper,[4] Damerau not only distinguished these four edit operations but also stated that they correspond to more than 80% of all human misspellings. Damerau's paper considered only misspellings that could be corrected with at most one edit operation.

The Damerau–Levenshtein distance differs from the classical Levenshtein distance by including transpositions among its allowable operations. The classical Levenshtein distance only allows insertion, deletion, and substitution operations.[5] Modifying this distance by including transpositions of adjacent symbols produces a different distance measure, known as the Damerau–Levenshtein distance.[2]

While the original motivation was to measure distance between human misspellings to improve applications such as spell checkers, Damerau–Levenshtein distance has also seen uses in biology to measure the variation between DNA.[6]

Contents [hide]

Definition [edit]

The Damerau–Levenshtein distance between two strings a and b is given by $d_{a,b}(|a|,|b|)$ where:

$$d_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} d_{a,b}(i-1,j)+1 \\ d_{a,b}(i,j-1)+1 \\ d_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \\ d_{a,b}(i-2,j-2)+1 \end{cases} & \text{if } i,j > 1 \text{ and } a_i = b_{j-1} \text{ and } a_{i-1} = b_j \\ \min \begin{cases} d_{a,b}(i-1,j)+1 \\ d_{a,b}(i,j-1)+1 \\ d_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

where $1_{(a_i \neq b_j)}$ is the indicator function equal to 0 when $a_i = b_j$ and equal to 1 otherwise.

Each recursive call matches one of the cases covered by the Damerau–Levenshtein distance:

- $d_{a,b}(i-1,j)+1$ corresponds to a deletion (from a to b).
- $d_{a,b}(i,j-1)+1$ corresponds to an insertion (from a to b).
- $d_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)}$ corresponds to a match or mismatch, depending on whether the respective symbols are the same.
- $d_{a,b}(i-2,j-2)+1$ corresponds to a transposition between two successive symbols.

Algorithm [edit]

Presented here are two algorithms: the first,[7] simpler one, computes what is known as the optimal string alignment[citation needed] (sometimes called the *restricted edit distance*[citation needed]), while the second one[8] computes the Damerau–Levenshtein distance with adjacent transpositions. Adding transpositions adds significant complexity. The difference between the two algorithms consists in that the *optimal string alignment algorithm* computes the number of edit operations needed to make the strings equal under the condition that **no substring is edited more than once**, whereas the second one presents no such restriction.

Take for example the edit distance between **CA** and **ABC**. The Damerau–Levenshtein distance LD(**CA**,**ABC**) = 2 because **CA** → **AC** → **ABC**, but the optimal string alignment distance OSA(**CA**,**ABC**) = 3 because if the operation **CA** → **AC** is used, it is not possible to use **AC** → **ABC** because that would require the substring to be edited more than once, which is not allowed in OSA, and therefore the shortest sequence of operations is **CA** → **A** → **AB** → **ABC**. Note that for the optimal string alignment distance, the triangle inequality does not hold: OSA(**CA**,**AC**) + OSA(**AC**,**ABC**) < OSA(**CA**,**ABC**), and so it is not a true metric.

Optimal string alignment distance [edit]

Firstly, let us consider a direct extension of the formula used to calculate Levenshtein distance. Below is pseudocode for a function *OptimalStringAlignmentDistance* that takes two strings, *str1* of length *lenStr1*, and *str2* of length *lenStr2*, and computes the optimal string alignment distance between them:

```
int OptimalStringAlignmentDistance(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
   // if implemented, 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],
                                d[i-2, j-2] + cost   // transposition
                             )

   return d[lenStr1, lenStr2]
```

Basically this is the algorithm to compute Levenshtein distance with one additional recurrence:

```
           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],
                                d[i-2, j-2] + cost   // transposition
                             )
```

Distance with adjacent transpositions [edit]

Here is the second algorithm that computes the true Damerau–Levenshtein distance with adjacent transpositions (ActionScript 3.0); this function requires as an additional parameter the size of the alphabet (*C*), so that all entries of the arrays are in 0..(*C*−1):

```
static public function damerauLevenshteinDistance(a:Array, b:Array, C:uint):uint
{
    // "infinite" distance is just the max possible distance
    var INF:uint = a.length + b.length;

    // make and initialize the character array indices
    var DA:Array = new Array(C);
    for (var k:uint = 0; k < C; ++k) DA[k]=0;

    // make the distance matrix H[-1..a.length][-1..b.length]
    var H:matrix = new matrix(a.length+2,b.length+2);

    // initialize the left and top edges of H
    H[-1][-1] = INF;
    for (var i:uint = 0; i <= a.length; ++i)
    {
        H[i][-1] = INF;
        H[i][ 0] = i;
    }
    for (var j:uint = 0; j <= b.length; ++j)
    {
        H[-1][j] = INF;
        H[ 0][j] = j;
    }

    // fill in the distance matrix H
    // look at each character in a
    for (var i:uint = 1; i <= a.length; ++i)
    {
        var DB:uint = 0;
        // look at each character in b
        for (var j:uint = 1; j <= b.length; ++j)
        {
            var i1:uint = DA[b[j-1]];
            var j1:uint = DB;
            var cost:uint;
            if (a[i-1] == b[j-1])
               {
                 cost = 0;
                 DB   = j;
               }
            else
               cost = 1;
            H[i][j] = Math.min(    H[i-1 ][j-1 ] + cost,  // substitution
                                   H[i   ][j-1 ] + 1,     // insertion
                                   H[i-1 ][j   ] + 1,     // deletion
                                   H[i1-1][j1-1] + (i-i1-1) + 1 + (j-j1-1));
        }
        DA[a[i-1]] = i;
    }
    return H[a.length][b.length];
}
```

Note: the algorithm given in the paper uses alphabet 1..C rather than the 0..*C*−1 used here; the paper indexes arrays: H[−1..|A|, −1..|B|] and DA[1..C]; here DA[0..C−1] is used; the paper seems to be missing the necessary line H[−1,−1] = INF

To devise a proper algorithm to calculate unrestricted Damerau–Levenshtein distance note that there always exists an optimal sequence of edit operations, where once-transposed letters are never modified afterwards. (This holds as long as the cost of a transposition, W_T, is at least the average of the cost of an insertion and deletion, i.e., $2W_T \geq W_I + W_D$.[8]) Thus, we need to consider only two symmetric ways of modifying a substring more than once: (1) transpose letters and insert an arbitrary number of characters between them, or (2) delete a sequence of characters and transpose letters that become adjacent after deletion. The straightforward implementation of this idea gives an algorithm of cubic complexity: $O\left(M \cdot N \cdot \max(M, N)\right)$, where *M* and *N* are string lengths. Using the ideas of Lowrance and Wagner,[8] this naive algorithm can be improved to be $O\left(M \cdot N\right)$ in the worst case.

It is interesting that the bitap algorithm can be modified to process transposition. See the information retrieval section of[1] for an example of such an adaptation.

Applications [edit]

Damerau–Levenshtein distance plays an important role in natural language processing. In natural languages, strings are short and the number of errors (misspellings) rarely exceeds 2. In such circumstances, restricted and real edit distance differ very rarely. Oommen and Loke[2] even mitigated the limitation of the restricted edit distance by introducing *generalized transpositions*. Nevertheless, one must remember that the restricted edit distance usually does not satisfy the triangle inequality and, thus, cannot be used with metric trees.

DNA [edit]

Since DNA frequently undergoes insertions, deletions, substitutions, and transpositions, and each of these operations occurs on approximately the same timescale, the Damerau–Levenshtein distance is an appropriate metric of the variation between two strands of DNA. More common in DNA, protein, and other bioinformatics related alignment tasks is the use of closely related algorithms such as Needleman–Wunsch algorithm or Smith–Waterman algorithm.

Fraud detection [edit]

The algorithm can be used with any set of words, like vendor names. Since entry is manual by nature there is a risk of entering a false vendor. A fraudster employee may enter one real vendor such as "Rich Heir Estate Services" versus a false vendor "Rich Hier State Services". The fraudster would then create a false bank account and have the company route checks to the real vendor and false vendor. The Damerau–Levenshtein algorithm will detect the transposed and dropped letter and bring attention of the items to a fraud examiner.

See also [edit]

- Approximate string matching
- Levenshtein automata
- Typosquatting

References [edit]

1. ^ Brill, Eric; Moore, Robert C. (2000). *An Improved Error Model for Noisy Channel Spelling Correction* (PDF). Proceedings of the 38th Annual Meeting on Association for Computational Linguistics. pp. 286–293. doi:10.3115/1075218.1075255.
2. ^ *a b* Bard, Gregory V. (2007), "Spelling-error tolerant, order-independent pass-phrases via the Damerau–Levenshtein string-edit distance metric", *Proceedings of the Fifth Australasian Symposium on ACSW Frontiers : 2007, Ballarat, Australia, January 30 - February 2, 2007*, Conferences in Research and Practice in Information Technology **68**, Darlinghurst, Australia: Australian Computer Society, Inc., pp. 117–124, ISBN 1-920-68285-6. The isbn produces two hits: a 2007 work and a 2010 work at World Cat.
3. ^ Li et al. (2006). *Exploring distributional similarity based models for query spelling correction* (PDF). Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics. pp. 1025–1032. doi:10.3115/1220175.1220304.
4. ^ Damerau, Fred J. (March 1964), "A technique for computer detection and correction of spelling errors", *Communications of the ACM* (ACM) **7** (3): 171–176, doi:10.1145/363958.363994
5. ^ Levenshtein, Vladimir I. (February 1966), "Binary codes capable of correcting deletions, insertions, and reversals", *Soviet Physics Doklady* **10** (8): 707–710
6. ^ The method used in: Majorek, Karolina A.; Dunin-Horkawicz, Stanisław et al. (2013), "The RNase H-like superfamily: new members, comparative structural analysis and evolutionary classification", *Nucleic Acids Research* **42** (7): 4160–4179, doi:10.1093/nar/gkt1414
7. ^ B. J. Oommen; R. K. S. Loke. "Pattern recognition of strings with substitutions, insertions, deletions and generalized transpositions". doi:10.1016/S0031-3203(96)00101-X. CiteSeerX: 10.1.1.50.1459.
8. ^ *a b c* Lowrance, Roy; Wagner, Robert A. (April 1975), "An Extension of the String-to-String Correction Problem", *JACM* **22** (2): 177–183, doi:10.1145/321879.321880

Further reading [edit]

- Navarro, Gonzalo (March 2001), "A guided tour to approximate string matching", *ACM Computing Surveys* **33** (1): 31–88, doi:10.1145/375360.375365

Categories: String similarity measures | Information theory | Dynamic programming