12. Edit Distance

String Alignment

Knuth, in (Knuth, 1993), shows how to transform **words** into **graph** using a seven step *ladder* of one letter substitutions.

Imagine applications where the distance between strings could be useful.

$$\text{words} \mapsto \text{wolds} \mapsto \text{golds} \mapsto \text{goads} \mapsto \text{grads} \mapsto \text{grade} \mapsto \text{grape} \mapsto \text{graph}.$$

Spell checkers provide a list of *nearby* words when a string is not found in the dictionary. DNA, the molecule of life, can be abstracted as strings over the alphabet

$$\mathbb{DNA} = \{A, C, G, T\}$$

Geneticists study the similarities and differences in DNA among members of a species and between species.

The similarity of two strings can be measured by an *edit distance*: The cost of a sequence of edit operations that change one string into another. Many different edit operations and cost measures have been proposed. Most measures define a *metric space*.

Definition 15: Metric on Strings

Let Σ^* be the set of all strings over an alphabet Σ. A metric d *is a function*

$$d : \Sigma^* \times \Sigma^* \mapsto [0, \infty)$$

with the properties:

1. ***Non-negativity:*** $d(s, t) \geq 0$ *for all strings* $s, t \in \Sigma^*$
2. ***Zero-distance:*** $d(s, t) = 0$ *if and only if* $s = t$
3. ***Symmetry:*** $d(s, t) = d(t, s)$
4. ***Triangle Inequality:*** $d(s, u) \leq d(s, t) + d(t, u)$

The common edit operations are *substitute*, *insert*, and *delete*.

Definition 16: String Operations

Let α and β be strings over Σ. Let x and y be a character in alphabet Σ. Define three operations:

- *Insertion:* $\alpha\beta \mapsto \alpha x\beta$
- *Deletion:* $\alpha x\beta \mapsto \alpha\beta$
- *Substitution:* $\alpha x\beta \mapsto \alpha y\beta$

For instance, to align $\alpha =$ ALGORITHM with $\beta =$ ALGEBRAIC this sequence of editing operations might be used, where s, i and d stand for *substitution*, *insertion*, and *deletion*.

A	L	G	O	-	R	-	I	T	H	M
A	L	G	E	B	R	A	I	C	-	-
			s	i		i		s	d	d

Is 6 the smallest unit cost distance between ALGORITHM and ALGEBRAIC?
Another metric charges 2 for a substitution.

If each operation has a cost of 1, then the distance between these two string is 6.

Levenshtein distance is a commonly used metric on strings. It is defined iteratively based on string length. Let λ denote the empty string. Then, the distance from any string α to λ is the length of α.

$$d(\alpha, \lambda) = d(\lambda, \alpha) = |\alpha|$$

Otherwise, let $d_{\alpha,\beta}(i, j)$ be the distance between $\alpha[1..i]$, the first i characters of α, and $\beta[1..j]$, the first j characters of β.

- If $\alpha[i]$ is deleted in aligning $\alpha[1..i]$ with $\beta[1..j]$, this edit distance is the distance between $\alpha[1..(i-1)]$ and $\beta[1..j]$ plus 1.
- If $\beta[j]$ is deleted in aligning $\alpha[1..i]$ with $\beta[1..j]$, this edit distance is the distance between $\alpha[1..i]$ and $\beta[1..(j-1)]$ plus 1.
- If $\alpha[i]$ is substituted $\beta[j]$ in aligning $\alpha[1..i]$ with $\beta[1..j]$, this edit distance is the distance between $\alpha[1..(i-1)]$ and $\beta[1..(j-1)]$ plus 1 if $\alpha[i] \neq \beta[j]$. And, otherwise, it is just $d((i-1), (j-1))$.

The value of $d(i, j)$ is the minimum over all of these values.

Definition 17: Levenshtein distance

Let α and β be strings over Σ. For $0 \leq i \leq |\alpha|$ and $0 \leq j \leq |\beta|$,

define the edit distance between $\alpha[1..i]$ *and* $\beta[1..j]$ *to be*

$$d(i, j) = \begin{cases} \max(i, j) & \textit{if } \min(i, j) = 0 \\ \min \begin{cases} d(i-1, j) + 1 \\ d(i, j-1) + 1 \\ d(i-1, j-1) + [\alpha_i \neq \beta_j] \end{cases} & \textit{otherwise} \end{cases}$$

where $[\texttt{False}] = 0$ *and* $[\texttt{True}] = 1$ *is the* characteristic *function.*

Problem 9: String Edit Distance Problem

Decision Problem: Given strings s *and* t*, is* m *the minimum number of edits to transform* s *into* t*?*
Function Problem: Given strings s *and* t*, find one or more edit sequences that minimize the distance between these strings.*

One alignment of

$$s = \text{TAGCTATCA} \quad \text{and} \quad t = \text{AGGCTATTA}$$

might look like this:

T	A	G	-	C	T	A	T	C	A
-	A	G	G	C	T	A	T	T	A
d			i					s	

The table below shows the initial configuration when computing the minimal edit distance between

$$s = \text{TAGCTATCA} \quad \text{and} \quad t = \text{AGGCTATTA}$$

The rows and columns are labeled by the characters in the strings.

The λ column shows the costs for inserting of TAGCTATCA into an empty string. These costs are $d(i, 0) = i$ for $i = 0, \ldots, 9$.

The λ row shows the costs for inserting AGGCTATTA into an empty string. These costs are $d(0, j) = j$ for $j = 0, \ldots, 9$.

	λ	A	G	G	C	T	A	T	T	A
λ	0	1	2	3	4	5	6	7	8	9
T	1									
A	2									
G	3									
C	4									
T	5									
A	6									
T	7									
C	8									
A	9									

Possible edits to compute $d(1, 1)$:

Substitute	
T	
A	
Delete–Insert	
T	A
-	A
Insert–Delete	
-	T
A	-

Values can be computed along diagonals. The first computed value comes from comparing T and A.

$$d(1, 1) = \begin{cases} d(0, 0)\ (= 0) & \text{if } T = A \\ \min \begin{cases} d(0, 1) + 1)\ (= 2) \\ d(1, 0) + 1)\ (= 2) \\ d(0, 0) + 1)\ (= 1) \end{cases} & \text{otherwise} \end{cases}$$

Downward moves $\downarrow$ and horizontal moves $\rightarrow$ always cost 1. Diagonal moves $\searrow$ cost 0 or 1 depending on whether the next characters match or not. The minimum of these costs is the new value.

	λ	A	G	G	C	T	A	T	T	A
λ	0	1	2	3	4	5	6	7	8	9
T	1	1								
A	2									
G	3									
C	4									
T	5									
A	6									
T	7									
C	8									
A	9									

Now, compute values in the next diagonal:

	λ	A	G	G	C	T	A	T	T	A
λ	0	1	2	3	4	5	6	7	8	9
T	1	1	2							
A	2	1								
G	3									
C	4									
T	5									
A	6									
T	7									
C	8									
A	9									

Next, compute $d(3, 1)$ $d(2, 2)$, and $d(1, 3)$

	λ	A	G	G	C	T	A	T	T	A
λ	0	1	2	3	4	5	6	7	8	9
T	1	1	3	3						
A	2	1	2							
G	3	2								
C	4									
T	5									
A	6									
T	7									
C	8									
A	9									

The complete edit distance table is:

	λ	A	G	G	C	T	A	T	T	A
λ	0	1	2	3	4	5	6	7	8	9
T	1	1	2	3	4	4	5	6	7	8
A	2	1	2	3	4	5	4	5	6	7
G	3	2	1	2	3	4	5	5	6	7
C	4	3	2	2	2	3	4	5	6	7
T	5	4	3	3	3	2	3	4	5	6
A	6	5	4	4	4	3	2	3	4	5
T	7	6	5	5	5	4	3	2	3	4
C	8	7	6	6	5	5	4	3	3	4
A	9	8	7	7	6	6	5	4	4	3

Tracing back in the array reveals optimal alignments.

	λ	A	G	G	C	T	A	T	T	A
λ	0	1	2	3	4	5	6	7	8	9
T	1	1	2	3	4	4	5	6	7	8
A	2	1	2	3	4	5	4	5	6	7
G	3	2	1	2	3	4	5	5	6	7
C	4	3	2	2	2	3	4	5	6	7
T	5	4	3	3	3	2	3	4	5	6
A	6	5	4	4	4	3	2	3	4	5
T	7	6	5	5	5	4	3	2	3	4
C	8	7	6	6	5	5	4	3	3	4
A	9	8	7	7	6	6	5	4	4	3

The optimal alignment has cost 3

T	A	G	-	C	T	A	T	C	A
-	A	G	G	C	T	A	T	T	A
d			i					s	

Here is a C implementation of edit distance.

Listing 58: Iterative String Edit Distance

169a ⟨*Iterative String Edit Distance* 169a⟩≡

```
int editDist(char *s, int ls, char *t, int lt)
{
  int distances[ls][lt];

  ⟨If either string is empty return 169b⟩
  ⟨Initialize the first row and first column 170a⟩
  ⟨For every pair of characters 170b⟩
  {
    ⟨If characters match, use the previous distance 170c⟩
    ⟨Otherwise, use the minimum distance 170d⟩
  }
  return distance[ls-1][lt-1];
}
```

If either string `s` or `t` is empty, return the length of the other, which translated to inserting its characters. The C-idiom is "if `ls=0`, then `!ls` is `True`."

169b ⟨*If either string is empty return* 169b⟩≡

```
if (!ls) return lt;
if (!lt) return ls;
```

Initializing the first row and column has time complexity $\Theta(n + m)$.

170a ⟨*Initialize the first row and first column* 170a⟩≡

```
for (int i = 0, int j = 0; i < m, j < n; i++, j++)
{
  distances[i][0] = i;
  distances[0][j] = j;
}
```

There are nm pairs of characters, assuming the source string `s` has length `ls = n` and target string `t` has length `lt = m`.

170b ⟨*For every pair of characters* 170b⟩≡

```
for (int i = 1; i < ls; i++)
  for (int j = 1; j < lt; j++)
```

Testing for a match has complexity $O(1)$.

170c ⟨*If characters match, use the previous distance* 170c⟩≡

```
if (s[i-1] == t[j-1]) {
  distance[i][j] = distance[i-1][j-1];
}
```

And when a mismatch occurs, only a few table look-ups, comparisons, and assignments are necessary.

170d ⟨*Otherwise, use the minimum distance* 170d⟩≡

```
else
{
  min = distance[i-1][j-1];
  if (min > distance[i][j-1])
  {
    min = distance[i][j-1];
  }
  if (min > distance[i-1][j])
  {
    min = distance[i-1][j];
  }
  distance[i][j] = 1 + min;
}
```

The performance of the edit distance algorithm is characterized by

- Time complexity: $O(nm)$ to account for the nested `for` loops.
- Space complexity: $O(nm)$ to account for storing the table.
- Trace-back: $O(n + m)$ to construct the optimal alignment.

Exercises

1. Fill in the the optimal alignment table for *seat* and *belt*.
2. Fill in the the optimal alignment table for *park* and *spake*.

Bibliography

Agrawal, M., Kayal, N., and Saxena, N. (2002). Primes is in P. http://www.cse.iitk.ac.in/news/primality.html. [page 205]

Aho, A. V. and Corasick, M. J. (1975). Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340. [page 87]

Apostolico, A. and Galil, Z. (1997). *Pattern Matching Algorithms*. Oxford University Press. [page 53]

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press. [page 141]

Bentley, J. (1984a). Programming pearls: Algorithm design techniques. *Commun. ACM*, 27(9):865–873. [page 33]

Bentley, J. (1984b). Programming Pearls: How to sort. *Commun. ACM*, 27(4):287–ff. [page 120]

Boyer, R. S. and Moore, J. S. (1977). A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772. [page 71]

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA. ACM. [page 199]

Corman, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, third edition. [page 9], [page 14], [page 18], [page 21], [page 31], [page 33], [page 39], [page 53], [page 93], [page 105], [page 135], [page 141], [page 163], [page 169], [page 178], [page 187], [page 193], [page 201]

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of Intractability*. W. H. Freeman. [page 193], [page 203]

Gödel, K. (1930). Die vollst andigkeit der axiome des logischen funktionenkalk uls. *Monatshefte f ur Mathematik und Physik*, 37:349–360. [page 194]

Gödel, K. (1992). *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. Dover books on advanced mathematics. Dover Publications. [page 194]

Gusfield, D. (1997). *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. EBL-Schweitzer. Cambridge University Press. [page 53]

Hoare, C. A. R. (1961). Quicksort. *Computer Journal*, 5(1):10–15. [page 119]

Hofstadter, D. R. (1999). *Gödel, Escher, Bach: An Eternal Golden Braid. 20th Anniversary Edition*. Basic Books, New York, New York. [page 25]

Kernighan, B. W. and Ritchie, D. M. (1988). *The* `C` *Programming Language*. Prentice Hall, second edition. [page 31]

Knuth, D. E. (1984). Literate programming. *The Computer Journal*, 27(2):97–111. [page 30]

Knuth, D. E. (1993). *The Stanford GraphBase: A Platform for Combinatorial Computing*. Addison-Wesley, ACM Press. [page 147], [page 163]

Knuth, D. E., Morris, J. H., and Pratt, V. R. (1977). Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):240–267. [page 55]

Lipovaca, M. (2011). *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, USA, 1st edition. [page 31]

Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley. [page 193], [page 195], [page 206]

Patterson, D. A. and Hennessy, J. L. (1996). *Computer Architecture: A Quantitive Approach*. Morgan Kaufmann Publishers, Inc., second edition. [page 30]

Polya, G. (1945). *How To Solve It*. Princeton University Press. ISBN 0-692-02356-5. [page 26]

Rabhi, F. A. and Lapalme, G. (1999). *Algorithms: A Functional Programming Approach*. Addison-Wesley. [page 31]

Ramsey, N. (1994). Literate programming simplified. *IEEE Software*, 11(5):97 – 105. [page 30]

Sankoff, D. and Kruskal, J. B., editors (1983). *Time Warps, String Edits, and Macromolecules*. Addison-Wesley, Reading, Massachusetts. [page 146]

Sedgewick, R. (1977). The analysis of quicksort programs. *Acta Informatica*, 7:327–355. [page 119]

Sedgewick, R. (1978). Implementing quicksort programs. *Communications of the ACM*, 21(10):847–857. [page 119]

Sedgewick, R. (2004). *Algorithms in Java*. Addison-Wesley. [page 31]

Shallit, J. (1994). Origins of the Analysis of the Euclidean Algorithm. *Historia Mathematica*, 21:401–419. [page 44]

Stansifer, R. (1984). Presburger's article on integer airthmetic: Remarks and translation. Technical Report TR84-639, Cornell University, Computer Science Department. [page 194]

Turing, A. M. (1936). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265. [page 197]

Wilf, H. (2006). *Generatingfunctionology: Third Edition*. AK Peters Series. Taylor & Francis. [page 19]

Index