Minimum Edit Distance

Definition of Minimum Edit Distance

- Many NLP tasks are concerned with measuring ***how similar two strings are***.

- Spell correction:
  - The user typed "graffe"
  - Which is closest? : **graf** **grail** **giraffe**
    - the word **giraffe**, which differs by only one letter from **graffe**, seems intuitively to be more similar than, say **grail** or **graf**,

- The **minimum edit distance** between two strings is defined as the ***minimum number of editing operations*** (**insertion**, **deletion**, **substitution**) needed to transform one string into another.

Minimum Edit Distance: Alignment

- The **minimum edit distance** between **`intention`** and **`execution`** can be visualized using their alignment.

- Given two sequences, an **alignment** is a correspondence between substrings of the two sequences.

```
I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
d s s   i s
```

Minimum Edit Distance

```
I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
d s s   i s
```

- If each operation has cost of 1
  - Distance between them is 5
- If substitutions cost 2 (**Levenshtein Distance**)
  - Distance between them is 8

Other uses of Edit Distance in NLP

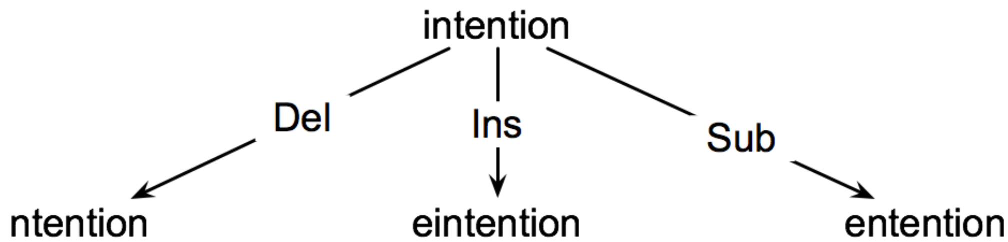
- Evaluating Machine Translation and speech recognition

```
R Spokesman confirms     senior government adviser was shot
H Spokesman said     the senior            adviser was shot dead
              S       I              D                        I
```

- Named Entity Extraction and Entity Coreference
  - IBM Inc. announced today
  - IBM profits
  - Stanford President John Hennessy announced yesterday
  - for Stanford University President John Hennessy

The Minimum Edit Distance Algorithm

- How do we find the minimum edit distance?
  - We can think of this as a **search task**, in which we are searching for **the shortest path**—a sequence of edits—from one string to another.



- The space of all possible edits is enormous, so we can't search naively.
  - Most of distinct edit paths ends up in the same state, so rather than recomputing all those paths, we could just remember ***the shortest path to a state*** each time we saw it.
  - We can do this by using **dynamic programming**.
  - **Dynamic programming** is the name for a class of algorithms that apply a table-driven method to solve problems by combining solutions to sub-problems.

Minimum Edit Distance between Two Strings

- For two strings
  - the source string X of length **n**
  - the target string Y of length **m**
- We define **D(i,j)** as the **edit distance** between X[1..i] and Y[1..j]
    - i.e., the first **i** characters of X and the first **j** characters of Y
- The **edit distance between X and Y** is thus **D(n,m)**

Dynamic Programming for Computing Minimum Edit Distance

- We will compute D(n,m) **bottom up**, combining solutions to subproblems.
- Compute **base cases** first:
  - D(i,0) = i
    - a source substring of length i and an empty target string requires i deletes.
  - D(0,j) = j
    - a target substring of length j and an empty source string requires j inserts.
- Having computed D(i,j) for small i, j we then compute larger D(i,j) based on previously computed smaller values.
- The value of D(i, j) is computed by taking the minimum of the three possible paths through the matrix which arrive there:

$$D[i,j] = \min \begin{cases} D[i-1,j] + \text{del-cost}(source[i]) \\ D[i,j-1] + \text{ins-cost}(target[j]) \\ D[i-1,j-1] + \text{sub-cost}(source[i], target[j]) \end{cases}$$

Dynamic Programming for Computing Minimum Edit Distance

- If we assume the version of **Levenshtein distance** in which the insertions and deletions each have a cost of 1, and substitutions have a cost of 2 (except substitution of identical letters have zero cost), the computation for D(i,j) becomes:

$$D[i,j] = \min \begin{cases} D[i-1,j]+1 \\ D[i,j-1]+1 \\ D[i-1,j-1] + \begin{cases} 2; & \text{if } source[i] \neq target[j] \\ 0; & \text{if } source[i] = target[j] \end{cases} \end{cases}$$

Minimum Edit Distance Algorithm

function MIN-EDIT-DISTANCE(*source*, *target*) **returns** *min-distance*

$n \leftarrow$ LENGTH(*source*)
$m \leftarrow$ LENGTH(*target*)
Create a distance matrix *distance[n+1,m+1]*

\# *Initialization: the zeroth row and column is the distance from the empty string*
$D[0,0] = 0$
for each row i **from** 1 **to** n **do**
$D[i,0] \leftarrow D[i\text{-}1,0] + \textit{del-cost}(\textit{source}[i])$
for each column j **from** 1 **to** m **do**
$D[0,j] \leftarrow D[0,j\text{-}1] + \textit{ins-cost}(\textit{target}[j])$

\# *Recurrence relation:*
for each row i **from** 1 **to** n **do**
for each column j **from** 1 **to** m **do**
$$D[i,j] \leftarrow \text{MIN}(\, D[i-1,j] + \textit{del-cost}(\textit{source}[i]),\; D[i-1,j-1] + \textit{sub-cost}(\textit{source}[i],\textit{target}[j]),\; D[i,j-1] + \textit{ins-cost}(\textit{target}[j]))$$

\# *Termination*
return D[n,m]

Computation of Minimum Edit Distance between intention and execution

N	9									
O	8									
I	7									
T	6									
N	5									
E	4									
T	3									
N	2									
I	1									
#	0	1	2	3	4	5	6	7	8	9
	#	E	X	E	C	U	T	I	O	N

Computation of Minimum Edit Distance between intention and execution

N	9									
O	8									
I	7									
T	6									
N	5									
E	4									
T	3									
N	2									
I	1									
#	0	1	2	3	4	5	6	7	8	9
	#	E	X	E	C	U	T	I	O	N

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 & \text{deletion} \\ D(i,j-1) + 1 & \text{insertion} \\ D(i-1,j-1) + \begin{cases} 2; & \text{if } S_1(i) \neq S_2(j) \\ 0; & \text{if } S_1(i) = S_2(j) \end{cases} & \text{substitution} \end{cases}$$

Computation of Minimum Edit Distance between intention and execution

N	9	8	9	10	11	12	11	10	9	8
O	8	7	8	9	10	11	10	9	8	9
I	7	6	7	8	9	10	9	8	9	10
T	6	5	6	7	8	9	8	9	10	11
N	5	4	5	6	7	8	9	10	11	10
E	4	3	4	5	6	7	8	9	10	9
T	3	4	5	6	7	8	7	8	9	8
N	2	3	4	5	6	7	8	7	8	7
I	1	2	3	4	5	6	7	6	7	8
#	0	1	2	3	4	5	6	7	8	9
	#	E	X	E	C	U	T	I	O	N

Computing Alignments

- Edit distance isn't sufficient
  - We often need to align each character of the two strings to each other
- We do this by keeping a "**backtrace**"
- Every time we enter a cell, remember where we came from
- When we reach the end,
  - Trace back the path from the upper right corner to read off the alignment

MinEdit with Backtrace

N	9									
O	8									
I	7									
T	6									
N	5									
E	4									
T	3									
N	2									
I	1									
#	0	1	2	3	4	5	6	7	8	9
	#	E	X	E	C	U	T	I	O	N

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 & \text{deletion} \\ D(i,j-1) + 1 & \text{insertion} \\ D(i-1,j-1) + \begin{cases} 2; & \text{if } S_1(i) \neq S_2(j) \\ 0; & \text{if } S_1(i) = S_2(j) \end{cases} & \text{substitution} \end{cases}$$

MinEdit with Backtrace

n	9	↓ 8	↙←↓ 9	↙←↓ 10	↙←↓ 11	↙←↓ 12	↓ 11	↓ 10	↓ 9	↙ **8**	
o	8	↓ 7	↙←↓ 8	↙←↓ 9	↙←↓ 10	↙←↓ 11	↓ 10	↓ 9	↙ **8**	← 9	
i	7	↓ 6	↙←↓ 7	↙←↓ 8	↙←↓ 9	↙←↓ 10	↓ 9	↙ **8**	← 9	← 10	
t	6	↓ 5	↙←↓ 6	↙←↓ 7	↙←↓ 8	↙←↓ 9	↙ **8**	← 9	← 10	←↓ 11	
n	5	↓ 4	↙←↓ 5	↙←↓ 6	↙←↓ 7	↙←↓ **8**	↙←↓ 9	↙←↓ 10	↙←↓ 11	↙↓ 10	
e	4	↙ 3	← 4	↙← **5**	← **6**	← 7	←↓ 8	↙←↓ 9	↙←↓ 10	↓ 9	
t	3	↙←↓ 4	↙←↓ **5**	↙←↓ 6	↙←↓ 7	↙←↓ 8	↙ 7	←↓ 8	↙←↓ 9	↓ 8	
n	2	↙←↓ **3**	↙←↓ 4	↙←↓ 5	↙←↓ 6	↙←↓ 7	↙←↓ 8	↓ 7	↙←↓ 8	↙ 7	
i	**1**	↙←↓ 2	↙←↓ 3	↙←↓ 4	↙←↓ 5	↙←↓ 6	↙←↓ 7	↙ 6	← 7	← 8	
#	**0**	1	2	3	4	5	6	7	8	9	
	#	**e**	**x**	**e**	**c**	**u**	**t**	**i**	**o**	**n**	

Adding Backtrace to Minimum Edit Distance

- Base conditions:

  `D(i,0) = i      D(0,j) = j`

  Termination:

  `D(N,M) is distance`

- Recurrence Relation:

```
For each  i = 1…M
     For each  j = 1…N
```

$$
D(i,j) = \min \begin{cases} D(i-1,j) + 1 & \text{deletion} \\ D(i,j-1) + 1 & \text{insertion} \\ D(i-1,j-1) + \begin{cases} 2; & \text{if } X(i) \neq Y(j) \\ 0; & \text{if } X(i) = Y(j) \end{cases} & \text{substitution} \end{cases}
$$

$$
ptr(i,j) = \begin{cases} \text{LEFT} & \text{insertion} \\ \text{DOWN} & \text{deletion} \\ \text{DIAG} & \text{substitution} \end{cases}
$$

Performance of Minimum Edit Distance Algorithm

- Time: O(nm)
- Space: O(nm)
- Backtrace: O(n+m)