Chapter 2: Corpus Processing Tools

Pierre Nugues

Lund University
Pierre.Nugues@cs.lth.se
http://cs.lth.se/pierre_nugues/
## Corpora

A corpus is a collection of texts (written or spoken) or speech. Corpora are balanced from different sources: news, novels, etc.

<table>
<thead>
<tr>
<th>Most frequent words in a collection of contemporary running texts</th>
<th>English</th>
<th>French</th>
<th>German</th>
</tr>
</thead>
<tbody>
<tr>
<td>the, of, to, in, and</td>
<td>the</td>
<td>de</td>
<td>der</td>
</tr>
<tr>
<td>Most frequent words in Genesis</td>
<td>and, the, of, his, he</td>
<td>et, de, la, à, il</td>
<td>und, die, der, da, er</td>
</tr>
</tbody>
</table>

- Most frequent words in a collection of contemporary running texts:
  - English: the, of, to, in, and
  - French: de, le (article), la (article), et, les
  - German: der, die, und, in, des

- Most frequent words in Genesis:
  - English: and, the, of, his, he
  - French: et, de, la, à, il
  - German: und, die, der, da, er
Characteristics of Current Corpora

Big: The Bank of English (Collins and U Birmingham) has more than 500 million words
Available in many languages
Easy to collect: The web is the largest corpus ever built and within the reach of a mouse click
Parallel: same text in two languages: English/French (Canadian Hansards), European parliament (23 languages)
Annotated with part-of-speech or manually parsed (treebanks):

- Characteristics/N of/PREP Current/ADJ Corpora/N
- (NP (NP Characteristics) (PP of (NP Current Corpora)))
Writing dictionaries
Dictionaries for language learners should be built on real usage

- *They’re just trying to score brownie points with politicians*
- *The boss is pleased – that’s another brownie point*

Bank of English: *brownie point* (6 occs) *brownie points* (76 occs)

Extensive use of corpora to:

- Find **concordances** and cite real examples
- Extract **collocations** and describe frequent pairs of words
Concordances

A word and its context:

<table>
<thead>
<tr>
<th>Language</th>
<th>Concordances</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>English</strong></td>
<td>s beginning of miracles did Je n they saw the miracles which n can do these miracles that t ain the second miracle that Je e they saw his miracles which</td>
</tr>
<tr>
<td><strong>French</strong></td>
<td>le premier des miracles que fi i dirent: Quel miracle nous mo om, voyant les miracles qu’il peut faire ces miracles que tu s ne voyez des miracles et des</td>
</tr>
</tbody>
</table>
Collocations

Word preferences: Words that occur together

<table>
<thead>
<tr>
<th></th>
<th>English</th>
<th>French</th>
<th>German</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>You say</strong></td>
<td>Strong tea</td>
<td>Thé fort</td>
<td>Schmales Gesicht</td>
</tr>
<tr>
<td></td>
<td>Powerful computer</td>
<td>Ordinateur puissant</td>
<td>Enge Kleidung</td>
</tr>
<tr>
<td><strong>You don’t say</strong></td>
<td>Strong computer</td>
<td>Thé puissant</td>
<td>Schmale Kleidung</td>
</tr>
<tr>
<td></td>
<td>Powerful tea</td>
<td>Ordinateur fort</td>
<td>Enges Gesicht</td>
</tr>
</tbody>
</table>
## Word Preferences

<table>
<thead>
<tr>
<th>Strong w</th>
<th>Powerful w</th>
</tr>
</thead>
<tbody>
<tr>
<td>strong w</td>
<td>powerful w</td>
</tr>
<tr>
<td>w</td>
<td></td>
</tr>
<tr>
<td>161</td>
<td>0</td>
</tr>
<tr>
<td>175</td>
<td>2</td>
</tr>
<tr>
<td>106</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Corpora as Knowledge Sources

Short term:

- Describe usage more accurately
- Assess tools: part-of-speech taggers, parsers.
- Learn statistical/machine learning models for speech recognition, taggers, parsers
- Derive automatically symbolic rules from annotated corpora

Longer term:

- Semantic processing
- Texts are the main repository of human knowledge
Finite-State Automata

A flexible tool to search and process text
A FSA accepts and generates strings, here $ac$, $abc$, $abbc$, $abbbbc$, $abbbbbbabbabc$, etc.
Mathematically defined by

- $Q$ a finite number of states;
- $\Sigma$ a finite set of symbols or characters: the input alphabet;
- $q_0$ a start state,
- $F$ a set of final states $F \subseteq Q$
- $\delta$ a transition function $Q \times \Sigma \rightarrow Q$ where $\delta(q, i)$ returns the state where the automaton moves when it is in state $q$ and consumes the input symbol $i$. 
% The start state
start(q0).

% The final states
final(q2).

transition(q0, a, q1).
transition(q1, b, q1).
transition(q1, c, q2).

accept(Symbols) :-
    start(StartState),
    accept(Symbols, StartState).

accept([], State) :-
    final(State).
accept([Symbol | Symbols], State) :-
    transition(State, Symbol, NextState),
    accept(Symbols, NextState).
Regular Expressions

Regexes are equivalent to FSA and generally easier to use

Constant regular expressions:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>regular</td>
<td>A section on regular expressions</td>
</tr>
<tr>
<td>the</td>
<td>The book of the life</td>
</tr>
</tbody>
</table>

The automaton above is described by the regex `ab*c`
grep 'ab*c' myFile1 myFile2
<table>
<thead>
<tr>
<th>Chars</th>
<th>Descriptions</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Matches any number of occurrences of the previous character – zero or more</td>
<td>ac*e matches strings ae, ace, acce, accce, etc. as in “The <strong>aerial</strong> acceleration alerted the <strong>ace</strong> pilot”</td>
</tr>
<tr>
<td>?</td>
<td>Matches at most one occurrence of the previous character – zero or one</td>
<td>ac?e matches ae and ace as in “The <strong>aerial</strong> acceleration alerted the <strong>ace</strong> pilot”</td>
</tr>
<tr>
<td>+</td>
<td>Matches one or more occurrences of the previous character</td>
<td>ac+e matches ace, acce, accce, etc. as in as in “The <strong>aerial</strong> acceleration alerted the <strong>ace</strong> pilot”</td>
</tr>
</tbody>
</table>
### Metacharacters

<table>
<thead>
<tr>
<th>Chars</th>
<th>Descriptions</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>{n}</td>
<td>Matches exactly ( n ) occurrences of the previous character</td>
<td>\texttt{ac{2}e} matches \texttt{acce} as in “The aerial acceleration alerted the ace pilot”</td>
</tr>
<tr>
<td>{n,}</td>
<td>Matches ( n ) or more occurrences of the previous character</td>
<td>\texttt{ac{2,}e} matches \texttt{acce}, \texttt{accce}, etc.</td>
</tr>
<tr>
<td>{n,m}</td>
<td>Matches from ( n ) to ( m ) occurrences of the previous character</td>
<td>\texttt{ac{2,4}e} matches \texttt{acce}, \texttt{accce}, and \texttt{acccce}.</td>
</tr>
</tbody>
</table>

Literal values of metacharacters must be quoted using \\. 
The Dot Metacharacter

The dot . is a metacharacter that matches one occurrence of any character except a new line.

a.e matches the strings ale and ace in:

The aerial acceleration alerted the ace pilot

as well as age, ape, are, ate, awe, axe, or aae, Aae, aBe, a1e, etc.

.* matches any string of characters until we encounter a new line.
The previous slide does not tell about the match strategy. Consider the string $aabbc$ and the regular expression $a+b*$.

By default the match engine is greedy: It matches as early and as many characters as possible and the result is $aabb$

Sometimes a problem. Consider the regular expression $<b>.*</b>$ and the phrase

$$They \text{ match } <b>\text{as early</b> and } <b>\text{as many</b> characters as they can.}$$

It is possible to use a lazy strategy with the *? metacharacter instead: $<b>.*?</b>$ and have the result:

$$They \text{ match } <b>\text{as early</b> and } <b>\text{as many</b> characters as they can.}$$
Character Classes

[... ] matches any character contained in the list.
[^...] matches any character not contained in the list.
[abc] means one occurrence of either a, b, or c
[^abc] means one occurrence of any character that is not an a, b, or c,
[ABCDEFGHIJKLMNOPQRSTUVWXYZ] one upper-case unaccented letter
[0123456789] means one digit.
[Cc]omputer [Ss]cience matches Computer Science,
computer Science, Computer science, computer science.
# Predefined Character Classes

<table>
<thead>
<tr>
<th>Expr.</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>\d</td>
<td>Any digit. Equivalent to [0–9]</td>
<td>A\dC matches A0C, A1C, A2C, A3C etc.</td>
</tr>
<tr>
<td>\D</td>
<td>Any nondigit. Equivalent to [^0–9]</td>
<td></td>
</tr>
<tr>
<td>\w</td>
<td>Any word character: letter, digit, or underscore.</td>
<td>1\w2 matches 1a2, 1A2, 1b2, 1B2, etc</td>
</tr>
<tr>
<td>\W</td>
<td>Any nonword character.</td>
<td></td>
</tr>
<tr>
<td>\s</td>
<td>Any white space character: space, tabulation, new line, form feed, etc.</td>
<td></td>
</tr>
<tr>
<td>\S</td>
<td>Any nonwhite space character.</td>
<td></td>
</tr>
</tbody>
</table>
## Nonprintable Symbols or Positions

<table>
<thead>
<tr>
<th>Char.</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>Matches the start of a line</td>
<td>^ab*c matches ac, abc, abbc, etc. when they are located at the beginning of a new line</td>
</tr>
<tr>
<td>$</td>
<td>Matches the end of a line</td>
<td>ab?c$ matches ac and abc when they are located at the end of a line</td>
</tr>
<tr>
<td>\b</td>
<td>Matches word boundaries</td>
<td>\babc matches abcd but not dabc</td>
</tr>
<tr>
<td>\n</td>
<td>Matches a new line</td>
<td>a\nb matches a</td>
</tr>
<tr>
<td>\t</td>
<td>Matches a tabulation</td>
<td>b</td>
</tr>
</tbody>
</table>
Union denoted $\mid$: $a \mid b$ means either $a$ or $b$.

Expression $a \mid bc$ matches the strings $a$ and $bc$ and $(a \mid b)c$ matches $ac$ and $bc$.

Order of precedence:

1. Closure and other repetition operator (highest)
2. Concatenation, line and word boundaries
3. Union (lowest)

$abc^*$ is the set $ab$, $abc$, $abcc$, $abccc$, etc.

$(abc)^*$ corresponds to $abc$, $abcabc$, $abcabcabc$, etc.
Perl

Match

```perl
while ($line = <>) {
    if ($line =~ m/ab+c/) {
        print $line;
    }
}
```

Substitute

```perl
while ($line = <>) {
    if ($line =~ m/ab+c/) {
        print "Old: ", $line;
        $line =~ s/ab+c/ABC/g;
        print "New: ", $line;
    }
}
```
Perl

**Translate**

```perl
tr/ABC/abc/
=line =~ tr/A-Z/a-z/;
=line =~ tr/AEIOUaeiou//d;
=line =~ tr/AEIOUaeiou/$/cs;
```

**Concatenate**

```perl
while ($line = <>) {
    $text .= $line;
}
print $text;
```

**References**

```perl
while ($line = <>) {
    while ($line =~ m/$ *[0-9]+)\.[0-9]*/g) {
        print "Dollars: ", $1, " Cents: ", $2, "\n"
    }
}
```
Predefined variables

```perl
$line = "Tell me, O muse, of that ingenious hero who travelled far and wide after he had sacked the famous town of Troy.";
$line =~ m/,.*,/;
print $&, "\n";
print "Before: ", $' , "\n";
print "After: ", $', "\n";
```

Arrays

```perl
@array = (1, 2, 3); #Array containing 1, 2, and 3
print $array[1]; #Prints 2
```
Concordances in Perl

# Modified from Doug Cooper
($file_name, $string, $width) = @ARGV;
open(FILE, "$file_name")
  || die "Could not open file $file_name.";
while ($line = <FILE>) {
  $text .= $line;
}
$string =~ s/ /\s/g; # spaces match tabs and new lines
$text =~ s/\n/ /g; # new lines are replaced by spaces
while ($text =~ m/(.{0,$width}$string.{0,$width})/g ) {
  # matches the pattern with 0..width to the right and left
  print "$1\n"; #$1 contains the match
}
Approximate String Matching

A set of edit operations that transforms a source string into a target string: copy, substitution, insertion, deletion, reversal (or transposition). Edits for *acress* from Kernighan et al. (1990).

<table>
<thead>
<tr>
<th>Typo</th>
<th>Correction</th>
<th>Source</th>
<th>Target</th>
<th>Position</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>acress</td>
<td>actress</td>
<td></td>
<td>t</td>
<td>2</td>
<td>Deletion</td>
</tr>
<tr>
<td>acress</td>
<td>cress</td>
<td>a</td>
<td></td>
<td>0</td>
<td>Insertion</td>
</tr>
<tr>
<td>acress</td>
<td>caress</td>
<td>ac</td>
<td>ca</td>
<td>0</td>
<td>Transposition</td>
</tr>
<tr>
<td>acress</td>
<td>access</td>
<td>r</td>
<td>c</td>
<td>2</td>
<td>Substitution</td>
</tr>
<tr>
<td>acress</td>
<td>across</td>
<td>e</td>
<td>o</td>
<td>3</td>
<td>Substitution</td>
</tr>
<tr>
<td>acress</td>
<td>acres</td>
<td>s</td>
<td></td>
<td>4</td>
<td>Insertion</td>
</tr>
<tr>
<td>acress</td>
<td>acres</td>
<td>s</td>
<td></td>
<td>5</td>
<td>Insertion</td>
</tr>
</tbody>
</table>
Minimum Edit Distance

Edit distances measure the similarity between strings. We compute the minimum edit distance using a matrix where the value at position \((i,j)\) is defined by the recursive formula:

\[
\text{edit}_\text{distance}(i, j) = \min \left( \begin{array}{c}
\text{edit}_\text{distance}(i-1, j) + \text{del}_\text{cost} \\
\text{edit}_\text{distance}(i-1, j-1) + \text{subst}_\text{cost} \\
\text{edit}_\text{distance}(i, j-1) + \text{ins}_\text{cost}
\end{array} \right).
\]

where \(\text{edit}_\text{distance}(i, 0) = i\) and \(\text{edit}_\text{distance}(0, j) = j\).
Edit Operations

Complementary to edit operations, edit distances measure the similarity between strings. They assign a cost to each edit operation, usually 0 to copies and 1 to deletions and insertions. Substitutions and transpositions correspond both to an insertion and a deletion. We can derive from this that they each have a cost of 2. Edit distances tell how far a source string is from a target string: the lower the distance, the closer the strings.

Given a set of edit operations, the minimum edit distance is the operation sequence that has the minimal cost needed to transform the source string into the target string. If we restrict the operations to copy/substitute, insert, and delete, we can represent the edit operations using a table, where the distance at a certain position in the table is derived from distances in adjacent positions already computed. This is expressed by the formula:

\[
\text{edit_distance}(i, j) = \min \begin{cases} \text{edit_distance}(i-1, j) + \text{del_cost} \\ \text{edit_distance}(i-1, j-1) + \text{subst_cost} \\ \text{edit_distance}(i, j-1) + \text{ins_cost} \end{cases}
\]

The boundary conditions for the first row and the first column correspond to a sequence of deletions and of insertions. They are defined as \( \text{edit_distance}(i, 0) = i \) and \( \text{edit_distance}(0, j) = j \).

We compute the cell values as a walk through the table from the beginning of the strings at the bottom left corner, and we proceed upward and rightward to fill adjacent cells from those where the value is already known.

Arrows in Fig. 2.10 represent the three edit operations, and Table 2.16 shows the distances to transform language into lineage. The value of the minimum edit distance is 5 and is shown at the upper right corner of the table.

![Fig. 2.10. Edit operations.](image)

The minimum edit distance algorithm is part of the dynamic programming techniques. Their principles are relatively simple. They use a table to represent data, and they solve a problem at a certain point by combining solutions to subproblems. Dynamic programming is a generic term that covers a set of widely used methods in optimization.

Usually, \( \text{del_cost} = \text{ins_cost} = 1 \)

\( \text{subst_cost} = 2 \) if \( \text{source}(i) \neq \text{target}(j) \)

\( \text{subst_cost} = 0 \) if \( \text{source}(i) = \text{target}(j) \).
Distance between $ab$ and $cb$

Let us align $a\ b$ Source

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c\ b$</td>
<td></td>
</tr>
</tbody>
</table>

- $b$ 2
- $c$ 1

Start 0 1 2

Start $a\ b$
Distance between $ab$ and $cb$

Let us align

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>c</td>
<td>b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>b</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Start</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Start</td>
<td>a</td>
</tr>
</tbody>
</table>

Pierre Nugues
An Introduction to Language Processing with Perl and Prolog
Distance between \( ab \) and \( cb \)

Let us align

\[
\begin{array}{ccc}
\text{Start} & \text{a} & \text{b} \\
\text{Source} & \text{b} & 2 & 3 \\
\text{c} & 1 & 2 & 3 \\
\text{Destination} & \text{Start} & \text{a} & \text{b} \\
\end{array}
\]
Distance between \( ab \) and \( cb \)

Let us align

\[
\begin{array}{c|c}
\text{a} & \text{b} \\
\hline
\text{c} & \text{b} \\
\end{array}
\]

Source

\[
\begin{array}{c|c|c|c}
\text{b} & 2 & 3 & 2 \\
\hline
\text{c} & 1 & 2 & 3 \\
\end{array}
\]

Destination

\[
\begin{array}{c|c|c}
\text{Start} & 0 & 1 & 2 \\
\hline
\text{Start} & \text{a} & \text{b} \\
\end{array}
\]
Distance between *language* and *lineage*

| e  | 7 |
| g  | 6 |
| a  | 5 |
| e  | 4 |
| n  | 3 |
| i  | 2 |
| l  | 1 |

<table>
<thead>
<tr>
<th>Start</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>l</td>
<td>a</td>
<td>n</td>
<td>g</td>
<td>u</td>
<td>a</td>
<td>g</td>
<td>e</td>
<td></td>
</tr>
</tbody>
</table>
Distance between *language* and *lineage*

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>6</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>a</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>e</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>n</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>i</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>l</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Start</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Start: l a n g u a g e
## Distance between *language* and *lineage*

<table>
<thead>
<tr>
<th></th>
<th>e</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>6</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>6</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>a</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>e</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>n</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>i</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>l</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Start</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>5</td>
</tr>
</tbody>
</table>

- Start: language
Perl Code

($source, $target) = @ARGV;
$length_s = length($source);
$length_t = length($target);
# Initialize first row and column
for ($i = 0; $i <= $length_s; $i++) {
    $table[$i][0] = $i;
}
for ($j = 0; $j <= $length_t; $j++) {
    $table[0][$j] = $j;
}
# Get the characters. Start index is 0
@source = split(//, $source);
@target = split(//, $target);
# Fills the table. Start index of rows and columns is 1
for ($i = 1; $i <= $length_s; $i++) {
    for ($j = 1; $j <= $length_t; $j++) {
        # Is it a copy or a substitution?
        $cost = ($source[$i-1] eq $target[$j-1]) ? 0 : 2;
        # Computes the minimum
        $min = $table[$i-1][$j-1] + $cost;
        if ($min > $table[$i][$j-1] + 1) {
            $min = $table[$i][$j-1] + 1;
        }
        if ($min > $table[$i-1][$j] + 1) {
            $min = $table[$i-1][$j] + 1;
        }
        $table[$i][$j] = $min;
    }
}
print "Minimum distance: ", $table[$length_s][$length_t], "\n";
% edit_distance(+Source, +Target, -Edits, ?Cost).
edit_distance(Source, Target, Edits, Cost) :-
    edit_distance(Source, Target, Edits, 0, Cost).
edit_distance([], [], [], Cost, Cost).
edit_distance(Source, Target, [EditOp | Edits], Cost, FinalCost) :-
    edit_operation(Source, Target, NewSource, NewTarget, EditOp, CostOp),
    Cost1 is Cost + CostOp,
    edit_distance(NewSource, NewTarget, Edits, Cost1, FinalCost).
% edit_operation carries out one edit operation
% between a source string and a target string.
edit_operation([Char | Source], [Char | Target], Source, Target, ident, 0).
edit_operation([SChar | Source], [TChar | Target], Source, Target, sub(SChar,TChar), 2) :-
    SChar \= TChar.
edit_operation([SChar | Source], Target, Source, Target, del(SChar), 1).
edit_operation(Source, [TChar | Target], Source, Target, ins(TChar), 1).
Distance between *language* and *lineage*

<table>
<thead>
<tr>
<th>Without epsilon symbols</th>
<th>First alignment</th>
<th>Third alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>language</td>
<td>language</td>
</tr>
<tr>
<td></td>
<td>/ / / /</td>
<td>/ / / /</td>
</tr>
<tr>
<td></td>
<td>lineage</td>
<td>lineage</td>
</tr>
<tr>
<td>With epsilon symbols</td>
<td>language</td>
<td>language</td>
</tr>
<tr>
<td></td>
<td>/ / / / / / / / / / / /</td>
<td>/ / / / / / / / / / / /</td>
</tr>
<tr>
<td></td>
<td>lineage</td>
<td>lineage</td>
</tr>
<tr>
<td></td>
<td>/ / / / / / / / / / / /</td>
<td>/ / / / / / / / / / / /</td>
</tr>
<tr>
<td></td>
<td>lineage</td>
<td>lineage</td>
</tr>
<tr>
<td></td>
<td>/ / / / / / / / / / / /</td>
<td>/ / / / / / / / / / / /</td>
</tr>
</tbody>
</table>
|                         | line ε age             | line ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε ε