Levenshtein distance

Eric Martin, CSE, UNSW

COMP9021 Principles of Programming

When typing a sequence of letters w that is not recognised as a syntactically correct word, corrections can be suggested in the form of one or more syntactically correct words w' "close enough" to w, that is, such that w can be converted into w' at a "minimal cost". The notion of *Levenshtein distance* offers a possible formalisation. One can convert a word w_1 into a word w_2 by successive applications of two transformations: *deletion* of an occurrence of a character and *insertion* of an occurrence of a character. For instance, one can convert DEPART into LEOPARD thanks to 2 deletions and 3 insertions, which can be achieved in many different ways, three of which are:

- DEPART -> EPART -> EPAR -> LEPAR -> LEOPAR -> LEOPARD
- DEPART -> LDEPART -> LDEOPART -> LDEOPARTD -> LEOPARTD -> LEOPARD
- DEPART -> EPART -> LEPART -> LEPAR -> LEPARD -> LEOPARD

Associate a cost to each of both transformations. Denote the cost of a deletion by C_δ and the cost of an insertion by C_ι. We arbitrarily set both C_δ and C_ι to 1, but will still reason, design and implement in terms of arbitrary values. The cost of a conversion that involves n_δ deletions and n_ι insertions is then defined as $n_\delta C_\delta + n_\iota C_\iota$, so $n_\delta + n_\iota$ with our choice of values for C_δ and C_ι. Given a word w, denote by $|w|$ the length of w. Of course the cost of converting a word w_1 into a word w_2 is at most equal to $|w_1|C_\delta + |w_2|C_\iota$: it suffices to delete all occurrences of characters in w_1 and insert all occurrences of characters in w_2. It is immediately verified that it is not possible to do any better if and only if w_1 and w_2 have no common character. We have seen that the cost of converting DEPART into LEOPARD is at most equal to $2C_\delta + 3C_\iota$, so 5 with our choice of values for C_δ and C_ι, and it is easy to verify that it is impossible to convert DEPART into LEOPARD at a lower cost.

We consider a third transformation: *substitution* of an occurrence of a character by an occurrence of another character, with an associated cost of C_ς. Since a deletion and an insertion achieve the same effect as a substitution, it is sensible to set C_ς to $C_\delta + C_\iota$, though other choices are legitimate. So DEPART can be converted into LEOPARD in new ways, still for a cost of 5 if $C_\delta = 1$, $C_\iota = 1$ and $C_\varsigma = C_\delta + C_\iota$, for instance with two substitutions and one insertion:

- DEPART -> LEPART -> LEPARD -> LEOPARD

The *Levenshtein distance* between two words w_1 and w_2 is the minimal cost of converting w_1 into w_2 using deletions, insertions and substitutions (allowing for substitutions makes no difference if $C_\varsigma = C_\delta + C_\iota$): it is the (unique) integer of the form $n_\delta C_\delta + n_\iota C_\iota + n_\varsigma C_\varsigma$ with (possibly nonunique) n_δ, n_ι and n_ς such that it is possible to convert w_1 into w_2 thanks to n_δ deletions, n_ι insertions and n_ς substitutions.

We are interested in computing the Levenshtein distance d between two words w_1 and w_2, and ignoring the order in which deletions, insertions and substitutions are applied, in finding all possible triples $(S_\delta, S_\iota, S_\varsigma)$ where S_δ is a set of occurrences of characters in w_1, S_ι is a set of occurrences of characters in w_2, and S_ς is a set of pairs consisting of an occurrence of a character in w_1 and an occurrence of a character in w_2, such that w_1 can be converted into w_2 by applying deletion, insertion and substitution to all members of S_δ, S_ι and S_ς, respectively, for a cost of d. Taking DEPART and LEOPARD as an example again, and with $C_\delta = 1$, $C_\iota = 1$ and $C_\varsigma = 2$, the Levenshtein distance between both words is 5 and there are 9 such triples, which can conveniently be represented as follows (underscores in the second word are aligned with members of S_δ, underscores in the first word are aligned with members of S_ι, and different letters aligned in both words correspond to members of S_ς):

- _DE_PAR_T
  L_EOPARD_
- D_E_PAR_T
  LEOPARD
- DE_PAR_T
  LEOPARD_
- _DE_PART_
  L_EOPAR_D
- D_E_PART_
  _LEOPAR_D
- DE_PART_
  LEOPAR_D
- _DE_PART
  L_EOPARD
- D_E_PART
  _LEOPARD
- DE_PART
  LEOPARD

There is only one way of minimal cost to convert a word w into the empty word: delete all occurrences of characters in w, for a cost of $|w|C_\delta$. Similarly, there is only one way of minimal cost to convert the empty word into a word w: insert all occurrences of characters in w, for a cost of $|w|C_\iota$. Consider two nonempty words w_1 and w_2. Write w_1 and w_2 as $w_1'c_1$ and $w_2'c_2$ with c_1 and c_2 the last characters in w_1 and w_2, respectively. In order to convert w_1 into w_2, one can

- either convert w_1' into w_2 and delete c_1,
- or convert w_1 into w_2' and insert c_2,
- or convert w_1' into w_2' and in case c_1 is distinct to c_2, substitute c_1 by c_2.

Let v_δ be C_δ plus the minimal cost of converting w_1' into w_2. Let v_ι be C_ι plus the minimal cost of converting w_1 into w_2'. Let v_ς be the minimal cost of converting w_1' into w_2' in case c_1 and c_2 are the same letters, and C_ς plus the minimal cost of converting w_1' into w_2' otherwise. Set $v = \min(v_\delta, v_\iota, v_\varsigma)$. Then v is the minimal cost of converting w_1 into w_2. Moreover,

- if $v = v_\delta$, all ways of converting w_1' into w_2 at minimal cost complemented with deleting c_1,
- if $v = v_\iota$, all ways of converting w_1 into w_2' at minimal cost complemented with inserting c_2,
- if $v = v_\varsigma$, all ways of converting w_1' into w_2' at minimal cost complemented with substituting c_1 by c_2 in case c_1 and c_2 are different

make up all ways of converting w_1 into w_2 at a minimal cost.

Given two words w_1 and w_2, consider the table with $|w_1| + 1$ many columns and $|w_2| + 1$ many rows defined as follows.

- For all $i \leq |w_1|$, the $(i+1)$-st column corresponds to the initial segment of w_1 of length i.
- For all $j \leq |w_2|$, the $(|w_2|-j+1)$-st row corresponds to the initial segment of w_2 of length j.
- For all $i \leq |w_1|$ and $j \leq |w_2|$, the element of the table at the intersection of the the $(i+1)$-st column and the $(|w_2|-j+1)$-st row records the minimal cost of converting the initial segment of w_1 of length i into the initial segment of w_2 of length j as well as
  - if $i > 0$, a horizontal arrow in case the minimal cost of converting the initial segment of w_1 of length i into the initial segment of w_2 of length j is equal to C_δ plus the minimal cost of converting the initial segment of w_1 of length $i-1$ into the initial segment of w_2 of length j,
  - if $j > 0$, a vertical arrow in case the minimal cost of converting the initial segment of w_1 of length i into the initial segment of w_2 of length j is equal to C_ι plus the minimal cost of converting the initial segment of w_1 of length i into the initial segment of w_2 of length $j-1$,
  - if $i > 0$ and $j > 0$, a diagonal arrow in case the minimal cost of converting the initial segment of w_1 of length i into the initial segment of w_2 of length j is equal to the minimal cost of converting the initial segment of w_1 of length $i-1$ into the initial segment of w_2 of length $j-1$ in case the $(i+1)$-st letter of w_1 and the $(j+1)$-st letter of w_2 are the same, and to C_ς plus the minimal cost of converting the initial segment of w_1 of length $i-1$ into the initial segment of w_2 of length $j-1$ otherwise.

For instance (with $C_\delta = 1$, $C_\iota = 1$ and $C_\varsigma = 2$), if w_1 is DEPART and w_2 is LEOPARD then (ignoring the red colour) the table can be depicted as:

7	D	7 $\downarrow$	6 $\swarrow$	$\leftarrow$ 7 $\downarrow$	6 $\downarrow$	5 $\downarrow$	4 $\downarrow$	$\leftarrow$ 5 $\swarrow$ $\downarrow$
6	R	6 $\downarrow$	$\leftarrow$ 7 $\swarrow$ $\downarrow$	6 $\downarrow$	5 $\downarrow$	4 $\downarrow$	3 $\swarrow$	$\leftarrow$ 4
5	A	5 $\downarrow$	$\leftarrow$ 6 $\swarrow$ $\downarrow$	5 $\downarrow$	4 $\downarrow$	3 $\swarrow$	$\leftarrow$ 4	$\leftarrow$ 5
4	P	4 $\downarrow$	$\leftarrow$ 5 $\swarrow$ $\downarrow$	4 $\downarrow$	3 $\swarrow$	$\leftarrow$ 4	$\leftarrow$ 5	$\leftarrow$ 6
3	O	3 $\downarrow$	$\leftarrow$ 4 $\swarrow$ $\downarrow$	3 $\downarrow$	$\leftarrow$ 4 $\swarrow$ $\downarrow$	$\leftarrow$ 5 $\swarrow$ $\downarrow$	$\leftarrow$ 6 $\swarrow$ $\downarrow$	$\leftarrow$ 7 $\swarrow$ $\downarrow$
2	E	2 $\downarrow$	$\leftarrow$ 3 $\swarrow$ $\downarrow$	2 $\swarrow$	$\leftarrow$ 3	$\leftarrow$ 4	$\leftarrow$ 5	$\leftarrow$ 6
1	L	1 $\downarrow$	$\leftarrow$ 2 $\swarrow$ $\downarrow$	$\leftarrow$ 3 $\swarrow$ $\downarrow$	$\leftarrow$ 4 $\swarrow$ $\downarrow$	$\leftarrow$ 5 $\swarrow$ $\downarrow$	$\leftarrow$ 6 $\swarrow$ $\downarrow$	$\leftarrow$ 7 $\swarrow$ $\downarrow$
0	·	0	$\leftarrow$ 1	$\leftarrow$ 2	$\leftarrow$ 3	$\leftarrow$ 4	$\leftarrow$ 5	$\leftarrow$ 6
		·	D	E	P	A	R	T
		0	1	2	3	4	5	6

For another example (still with $C_\delta = 1$, $C_\iota = 1$ and $C_\varsigma = 2$), if w_1 is PAPER and w_2 is POPE then (ignoring the red colour) the table can be depicted as:

4	E	4 ↓	3 ↓	← 4 ↙ ↓	3 ↑	2 ↙	← 3
3	P	3 ↓	2 ↙ ↓	← 3 ↙ ↓	2 ↙	← 3	← 4
2	O	2 ↓	1 ↓	← 2 ↙ ↑	← 3 ↙ ↑	← 4 ↙ ↑	← 5 ↙ ↑
1	P	1 ↓	0 ↙	← 1	← 2 ↙	← 3	← 4
0	·	0	← 1	← 2	← 3	← 4	← 5
		·	P	A	P	E	R
		0	1	2	3	4	5

The following observations derive from the previous considerations.

- For all $i \leq |w_1|$, the $(i+1)$-st column of the $(|w_2|+1)$-st row records i and a horizontal arrow.
- For all $j \leq |w_2|$, the $(|w_2|-j+1)$-st row of the first column records j and a vertical arrow.
- An element inside the table can be determined if the element to the left, the element below, and the element to the left and below have all been determined, hence it is possible to build the table by proceeding from the lower left corner up to the upper right corner.
- The Levenshtein distance between w_1 and w_2 is recorded in the table's top right corner.
- All possible ways of converting w_1 into w_2 at a minimal cost correspond to all paths from the lower left corner up to the upper right corner, which are best identified starting from the upper right corner, following the arrows left, down and diagonally; with the previous two table examples, there are 9 and 6 such paths, respectively, easily identified from the red arrows.

We will eventually define a class `Levenshtein_distance` but we first illustrate its functionality with tracing functions, using the DEPART -> LEOPARD example and with the suggested costs for deletion, insertion and substitution:

[1]:
```
deletion_cost = 1
insertion_cost = 1
substitution_cost = 2

word_1 = 'DEPART'
word_2 = 'LEOPARD'
```

The table that has been described above is implemented as a list of lists, best viewed as a sequence of columns, read from left to right, each column being read from bottom to top, with DEPART and LEOPARD positioned as follows:

D
R
A
P
O
E
L
.
    . D E P A R T

Each member of this list of lists is a pair whose first element is the minimal cost of converting the corresponding initial segment of DEPART with the corresponding initial segment of LEOPARD, and whose second element is a string containing a direction, that is:

- the character '/' if a diagonal move allows one to yield this minimal cost, thanks to a match or a substitution of the last characters of both initial segments;
- the character '-' if a horizontal move allows one to yield this minimal cost, thanks to a deletion of the last character of the initial segment of DEPART;
- the character '|' if a vertical move allows one to yield this minimal cost, thanks to an insertion of the last character of the initial segment of LEOPARD.

We also define a function to display the table, which to start with, stores costs of 0 and no move in all cells:

[2]:
```
N_1 = len(word_1) + 1
N_2 = len(word_2) + 1
table = [[(0, []) for _ in range(N_2)] for _ in range(N_1)]

def represent(cell):
    return ''.join(x in cell[1] and x or ' ' for x in '/-|') + str(cell[0])

def display_table():
    for j in range(1, N_2):
        print(N_2 - j, word_2[N_2 - j - 1], end=' ')
        print(' '.join(represent(table[i][j]) for i in range(N_1)))
    print('0 .', ' '.join(represent(table[i][0]) for i in range(N_1)))
    print('       .   ', '    '.join(word_1[i] for i in range(N_1 - 1)))
    print('       0   ', '    '.join(str(i) for i in range(1, N_1)))

display_table()
```

```
7 D    0    0    0    0    0    0    0
6 R    0    0    0    0    0    0    0
5 A    0    0    0    0    0    0    0
4 P    0    0    0    0    0    0    0
3 O    0    0    0    0    0    0    0
2 E    0    0    0    0    0    0    0
1 L    0    0    0    0    0    0    0
0 .    0    0    0    0    0    0    0
       .    D    E    P    A    R    T
       0    1    2    3    4    5    6
```

We have seen that it is straightforward to fill the first column and the last row of the table:

[3]:
```
for i in range(1, N_1):
    table[i][0] = i, ['-']
for j in range(1, N_2):
    table[0][j] = j, ['|']
```

```
display_table()
```

```
7 D   |1    0    0    0    0    0    0
6 R   |2    0    0    0    0    0    0
5 A   |3    0    0    0    0    0    0
4 P   |4    0    0    0    0    0    0
3 O   |5    0    0    0    0    0    0
2 E   |6    0    0    0    0    0    0
1 L   |7    0    0    0    0    0    0
0 .    0  - 1  - 2  - 3  - 4  - 5  - 6
       .    D    E    P    A    R    T
       0    1    2    3    4    5    6
```

We then fill all cells of the table column by column, from left to right, from bottom to top for a given column:

[4]:
```
d = {}
for i in range(1, N_1):
    for j in range(1, N_2):
        d['-'] = table[i - 1][j][0] + deletion_cost
        d['|'] = table[i][j - 1][0] + insertion_cost
        d['/'] = table[i - 1][j - 1][0] if word_1[i - 1] == word_2[j - 1]\
                     else table[i - 1][j - 1][0] + substitution_cost
        minimal_cost = min(d.values())
        table[i][j] = minimal_cost, [x for x in d if d[x] == minimal_cost]
    display_table()
    print()
```

```
7 D   |1 /-|2    0    0    0    0    0
6 R   |2 /-|3    0    0    0    0    0
5 A   |3 /-|4    0    0    0    0    0
4 P   |4 /-|5    0    0    0    0    0
3 O   |5 /-|6    0    0    0    0    0
2 E   |6 /-|7    0    0    0    0    0
1 L   |7 /   6    0    0    0    0    0
0 .    0  - 1  - 2  - 3  - 4  - 5  - 6
       .    D    E    P    A    R    T
       0    1    2    3    4    5    6

7 D   |1 /-|2 /-|3    0    0    0    0
6 R   |2 /-|3 /   2    0    0    0    0
5 A   |3 /-|4   |3    0    0    0    0
4 P   |4 /-|5   |4    0    0    0    0
3 O   |5 /-|6   |5    0    0    0    0
2 E   |6 /-|7   |6    0    0    0    0
1 L   |7 /   6  -|7    0    0    0    0
0 .    0  - 1  - 2  - 3  - 4  - 5  - 6
       .    D    E    P    A    R    T
```

```
       0    1    2    3    4    5    6

7 D   |1 /-|2 /-|3 /-|4    0    0    0
6 R   |2 /-|3 /  2  - 3    0    0    0
5 A   |3 /-|4    |3 /-|4    0    0    0
4 P   |4 /-|5    |4 /  3    0    0    0
3 O   |5 /-|6    |5    |4    0    0    0
2 E   |6 /-|7    |6    |5    0    0    0
1 L   |7 /  6  -|7    |6    0    0    0
0 .    0  - 1  - 2  - 3  - 4  - 5  - 6
       .    D    E    P    A    R    T
       0    1    2    3    4    5    6

7 D   |1 /-|2 /-|3 /-|4 /-|5    0    0
6 R   |2 /-|3 /  2  - 3  - 4    0    0
5 A   |3 /-|4    |3 /-|4 /-|5    0    0
4 P   |4 /-|5    |4 /  3  - 4    0    0
3 O   |5 /-|6    |5    |4 /  3    0    0
2 E   |6 /-|7    |6    |5    |4    0    0
1 L   |7 /  6  -|7    |6    |5    0    0
0 .    0  - 1  - 2  - 3  - 4  - 5  - 6
       .    D    E    P    A    R    T
       0    1    2    3    4    5    6

7 D   |1 /-|2 /-|3 /-|4 /-|5 /-|6    0
6 R   |2 /-|3 /  2  - 3  - 4  - 5    0
5 A   |3 /-|4    |3 /-|4 /-|5 /-|6    0
4 P   |4 /-|5    |4 /  3  - 4  - 5    0
3 O   |5 /-|6    |5    |4 /  3  - 4    0
2 E   |6 /-|7    |6    |5    |4 /  3    0
1 L   |7 /  6  -|7    |6    |5    |4    0
0 .    0  - 1  - 2  - 3  - 4  - 5  - 6
       .    D    E    P    A    R    T
       0    1    2    3    4    5    6

7 D   |1 /-|2 /-|3 /-|4 /-|5 /-|6 /-|7
6 R   |2 /-|3 /  2  - 3  - 4  - 5  - 6
5 A   |3 /-|4    |3 /-|4 /-|5 /-|6 /-|7
4 P   |4 /-|5    |4 /  3  - 4  - 5  - 6
3 O   |5 /-|6    |5    |4 /  3  - 4  - 5
2 E   |6 /-|7    |6    |5    |4 /  3  - 4
1 L   |7 /  6  -|7    |6    |5    |4 /-|5
0 .    0  - 1  - 2  - 3  - 4  - 5  - 6
       .    D    E    P    A    R    T
       0    1    2    3    4    5    6
```

The Levenshtein distance is to be found in the top right corner of the table:

[5]:
```
def distance():
    return table[len(word_1)][len(word_2)][0]

distance()
```

[5]: 5

To compute all ways of converting DEPART into LEOPARD at a minimal cost, we define a generator function that operates on `backtraces`, the projection of `table` that for each cell, discards the first member (the cost) and keeps the second member (the directions). We trace execution, indicating whether a substitution (/), a match (*), a deletion (-) or an insertion (|) is performed, indicating the row (`j`) and column (`i`) indexes of the cell where that match or transformation is done, and showing the result of the transformations done so far on both initial segments of DEPART and LEOPARD determined by that cell. It demonstrates how each of the nine paths that join the bottom left corner to the top right corner of the table are retrieved:

[6]:
```
backtraces = [[table[i][j][1] for j in range(len(word_2) + 1)
              ] for i in range(len(word_1) + 1)
             ]

def compute_alignments(i, j):
    if i == j == 0:
        yield '', ''
    if '/' in backtraces[i][j]:
        for pair in compute_alignments(i - 1, j - 1):
            if word_1[i - 1] == word_2[j - 1]:
                print(f'{"  " * i}*  i = {i} j = {j} ',
                      pair[0] + word_1[i - 1], pair[1] + word_2[j - 1]
                     )
            else:
                print(f'{"  " * i}/  i = {i} j = {j} ',
                      pair[0] + word_1[i - 1], pair[1] + word_2[j - 1]
                     )
            yield pair[0] + word_1[i - 1], pair[1] + word_2[j - 1]
    if '-' in backtraces[i][j]:
        for pair in compute_alignments(i - 1, j):
            print(f'{"  " * i}-  i = {i} j = {j} ', pair[0] + word_1[i - 1],
                  pair[1] + '_'
                 )
            yield pair[0] + word_1[i - 1], pair[1] + '_'
    if '|' in backtraces[i][j]:
        for pair in compute_alignments(i, j - 1):
            print(f'{"  " * i}|  i = {i} j = {j} ', pair[0] + '_',
                  pair[1] + word_2[j - 1]
                 )
            yield pair[0] + '_', pair[1] + word_2[j - 1]
```

```python
list(compute_alignments(len(word_1), len(word_2)))
```

```
  /  i = 1 j = 1  D L
    *  i = 2 j = 2  DE LE
    |  i = 2 j = 3  DE_ LEO
      *  i = 3 j = 4  DE_P LEOP
        *  i = 4 j = 5  DE_PA LEOPA
          *  i = 5 j = 6  DE_PAR LEOPAR
            /  i = 6 j = 7  DE_PART LEOPARD
|  i = 0 j = 1  _ L
  -  i = 1 j = 1  _D L_
    *  i = 2 j = 2  _DE L_E
    |  i = 2 j = 3  _DE_ L_EO
      *  i = 3 j = 4  _DE_P L_EOP
        *  i = 4 j = 5  _DE_PA L_EOPA
          *  i = 5 j = 6  _DE_PAR L_EOPAR
            /  i = 6 j = 7  _DE_PART L_EOPARD
  -  i = 1 j = 0  D _
  |  i = 1 j = 1  D_ _L
    *  i = 2 j = 2  D_E _LE
    |  i = 2 j = 3  D_E_ _LEO
      *  i = 3 j = 4  D_E_P _LEOP
        *  i = 4 j = 5  D_E_PA _LEOPA
          *  i = 5 j = 6  D_E_PAR _LEOPAR
            /  i = 6 j = 7  D_E_PART _LEOPARD
  /  i = 1 j = 1  D L
    *  i = 2 j = 2  DE LE
    |  i = 2 j = 3  DE_ LEO
      *  i = 3 j = 4  DE_P LEOP
        *  i = 4 j = 5  DE_PA LEOPA
          *  i = 5 j = 6  DE_PAR LEOPAR
          |  i = 5 j = 7  DE_PAR_ LEOPARD
            -  i = 6 j = 7  DE_PAR_T LEOPARD_
|  i = 0 j = 1  _ L
  -  i = 1 j = 1  _D L_
    *  i = 2 j = 2  _DE L_E
    |  i = 2 j = 3  _DE_ L_EO
      *  i = 3 j = 4  _DE_P L_EOP
        *  i = 4 j = 5  _DE_PA L_EOPA
          *  i = 5 j = 6  _DE_PAR L_EOPAR
          |  i = 5 j = 7  _DE_PAR_ L_EOPARD
            -  i = 6 j = 7  _DE_PAR_T L_EOPARD_
  -  i = 1 j = 0  D _
  |  i = 1 j = 1  D_ _L
    *  i = 2 j = 2  D_E _LE
    |  i = 2 j = 3  D_E_ _LEO
      *  i = 3 j = 4  D_E_P _LEOP
```

```
        *  i = 4 j = 5  D_E_PA _LEOPA
          *  i = 5 j = 6  D_E_PAR _LEOPAR
          |  i = 5 j = 7  D_E_PAR_ _LEOPARD
            -  i = 6 j = 7  D_E_PAR_T _LEOPARD_
  /  i = 1 j = 1  D L
    *  i = 2 j = 2  DE LE
    |  i = 2 j = 3  DE_ LEO
      *  i = 3 j = 4  DE_P LEOP
        *  i = 4 j = 5  DE_PA LEOPA
          *  i = 5 j = 6  DE_PAR LEOPAR
            -  i = 6 j = 6  DE_PART LEOPAR_
            |  i = 6 j = 7  DE_PART_ LEOPAR_D
|  i = 0 j = 1  _ L
  -  i = 1 j = 1  _D L_
    *  i = 2 j = 2  _DE L_E
    |  i = 2 j = 3  _DE_ L_EO
      *  i = 3 j = 4  _DE_P L_EOP
        *  i = 4 j = 5  _DE_PA L_EOPA
          *  i = 5 j = 6  _DE_PAR L_EOPAR
            -  i = 6 j = 6  _DE_PART L_EOPAR_
            |  i = 6 j = 7  _DE_PART_ L_EOPAR_D
  -  i = 1 j = 0  D _
  |  i = 1 j = 1  D_ _L
    *  i = 2 j = 2  D_E _LE
    |  i = 2 j = 3  D_E_ _LEO
      *  i = 3 j = 4  D_E_P _LEOP
        *  i = 4 j = 5  D_E_PA _LEOPA
          *  i = 5 j = 6  D_E_PAR _LEOPAR
            -  i = 6 j = 6  D_E_PART _LEOPAR_
            |  i = 6 j = 7  D_E_PART_ _LEOPAR_D
```

[6]:
```
[('DE_PART', 'LEOPARD'),
 ('_DE_PART', 'L_EOPARD'),
 ('D_E_PART', '_LEOPARD'),
 ('DE_PAR_T', 'LEOPARD_'),
 ('_DE_PAR_T', 'L_EOPARD_'),
 ('D_E_PAR_T', '_LEOPARD_'),
 ('DE_PART_', 'LEOPAR_D'),
 ('_DE_PART_', 'L_EOPAR_D'),
 ('D_E_PART_', '_LEOPAR_D')]
```

Putting everything together in a class, with an extra display method, `display_all_aligned_pairs()`:

[7]:
```python
class Levenshtein_distance:
    def __init__(self, word_1, word_2, insertion_cost=1, deletion_cost=1,
                 substitution_cost=2
                ):
```

```python
        self.word_1 = word_1
        self.word_2 = word_2
        self.insertion_cost = insertion_cost
        self.deletion_cost = deletion_cost
        self.substitution_cost = substitution_cost
        self._table = self._get_distances_and_backtraces_table()
        self._backtraces = [[self._table[i][j][1]
                                 for j in range(len(self.word_2) + 1)
                            ] for i in range(len(self.word_1) + 1)
                           ]
        self.aligned_pairs = self.get_aligned_pairs()

    def _get_distances_and_backtraces_table(self):
        N_1 = len(self.word_1) + 1
        N_2 = len(self.word_2) + 1
        table = [[(0, []) for _ in range(N_2)] for _ in range(N_1)]
        for i in range(1, N_1):
            table[i][0] = i, ['-']
        for j in range(1, N_2):
            table[0][j] = j, ['|']
        d = {}
        for i in range(1, N_1):
            for j in range(1, N_2):
                d['-'] = table[i - 1][j][0] + self.deletion_cost
                d['|'] = table[i][j - 1][0] + self.insertion_cost
                d['/'] = table[i - 1][j - 1][0]\
                          if self.word_1[i - 1] == self.word_2[j - 1]\
                          else table[i - 1][j - 1][0] + self.substitution_cost
                minimal_cost = min(d.values())
                table[i][j] = minimal_cost,\
                              [x for x in d if d[x] == minimal_cost]
        return table

    def _compute_alignments(self, i, j):
        if i == j == 0:
            yield '', ''
        if '/' in self._backtraces[i][j]:
            for pair in self._compute_alignments(i - 1, j - 1):
                yield pair[0] + self.word_1[i - 1],\
                      pair[1] + self.word_2[j - 1]
        if '-' in self._backtraces[i][j]:
            for pair in self._compute_alignments(i - 1, j):
                yield pair[0] + self.word_1[i - 1], pair[1] + '_'
        if '|' in self._backtraces[i][j]:
            for pair in self._compute_alignments(i, j - 1):
                yield pair[0] + '_', pair[1] + self.word_2[j - 1]
```

```python
    def distance(self):
        return self._table[len(self.word_1)][len(self.word_2)][0]

    def get_aligned_pairs(self):
        return list(self._compute_alignments(len(self.word_1),
                                             len(self.word_2)
                                            )
                   )

    def display_all_aligned_pairs(self):
        print('\n\n'.join('\n'.join((pair[0], pair[1]))
                          for pair in self.aligned_pairs
                         )
             )

ld = Levenshtein_distance('DEPART', 'LEOPARD')
ld.distance()
ld.get_aligned_pairs()
ld.display_all_aligned_pairs()
```

[7]: 5

[7]:
```
[('DE_PART', 'LEOPARD'),
 ('_DE_PART', 'L_EOPARD'),
 ('D_E_PART', '_LEOPARD'),
 ('DE_PAR_T', 'LEOPARD_'),
 ('_DE_PAR_T', 'L_EOPARD_'),
 ('D_E_PAR_T', '_LEOPARD_'),
 ('DE_PART_', 'LEOPAR_D'),
 ('_DE_PART_', 'L_EOPAR_D'),
 ('D_E_PART_', '_LEOPAR_D')]
```

```
DE_PART
LEOPARD

_DE_PART
L_EOPARD

D_E_PART
_LEOPARD

DE_PAR_T
LEOPARD_

_DE_PAR_T
L_EOPARD_
```

```
D_E_PAR_T
_LEOPARD_

DE_PART_
LEOPAR_D

_DE_PART_
L_EOPAR_D

D_E_PART_
_LEOPAR_D
```