Introduction to algorithmic Fall 2017

IFT2125-6001 TA: Maëlle Zimmermann

Demonstration 11

1

Question: Give an algorithm that calculates the editing distance d, also called
distance from Levenshtein, between two words. More precisely, let u and v be two words,
then d (u, v) is the minimum number of letters to delete, insert or substitute
to go from u to v

For example, we can go from table to stage in 3 operations:

table → tale	(delete b)
→ stale	(insert s)
→ stage	(substitute l for g)

Solution: We are building an array D of size | u | + 1 × | v | + 1 such that D [i] [j] is
the distance between the subwords u [1 ... i] and v [1 ... j].

Let us first note that D [i] [0] = i for all $0 \leq i \leq |u|$, and D [0] [j] = j for all $0 \leq j \leq |v|$.
Indeed, for an empty word ε, we have ad (x, ε) = d (ε, x) = | x | because the minimal edition consists of
in the first case to erase all the letters of x (| x | deletions) and in the second
where to add all the letters of x to the empty word (| x | insertions). More generally, for
go from a word u [1 ... i] to a word v [1 ... j], there are three possibilities:

- Delete u [i] then go from u [1 ... i - 1] to av [1 ... j]
- Change from u [1 ... i] to av [1 ... j - 1] and insert v [j]
- Change from u [1 ... i - 1] to av [1 ... j - 1] and substitute u [i] for v [j] (if these letters are different)

This gives the rule:

$$D[i][j] = \min(D[i-1][j] + 1 \text{ (suppression)}, D[i][j-1] + 1 \text{ (insertion)}, D[i-1][j-1] + b_{i,j} \text{ (substitution)})$$

o`ub $_{i,j}$ = 0 if u [i] = v [j] and 1 otherwise.

Here is an implementation of the algorithm, which takes a time in Θ (| u || v |):

```
def d_tab (u, v):
    D = [[0] * (len (v) + 1) for in range (len (u) + 1)]

    for i in range (len (u) +1):
        D [i] [0] = i

    for j in range (len (v) +1):
        D [0] [j] = j

    for i in range (1, len (u) + 1):
        for j in range (1, len (v) + 1):
            b = 1 if u [i]! = v [j] else 0
            D [i] [j] = min (D [i-1] [j] + 1, D [i] [j-1] + 1, D [i-1] [j-1] + b)
    return D

def d (u, v):
    return d_tab (u, v) [- 1] [- 1]
```

2

Question: Let u = AGGAGGA and v = AAGCTAAG. Identify all alignments between u and v of cost d (u, v), that is to say all the optimal alignments.

Solution: The distances table is:

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

So the Levenshtein distance between the two words is 5. to find which line
and therefore what sequence of operations gives this cost, we must go back the table
from the box T [-1] [- 1] following the rules of the algorithm.

The top path corresponds to the following sequence alignment:

```
AAGCTAAG                    -
- AG          - GAGGA
```

which corresponds to making two insertions (A and C), two substitutions (G for T and
G for A) and a deletion (A). In all we get 6 paths, and the other 5
correspond to the following sequence alignments:

```
AAGCTAAG
AGGAGGA                     -

AAGCTAAG                    -
AGG          - - AGGA

AAGCTAAG                    -
AT   - G       - GAGGA

AAGCTAAG                    -
- AGG            - AGGA

AAGCTAAG                    -
AT   - GG          - AGGA
```

3

Question: Make a thorough journey on the following graph and find its points hinge:

Solution: Here is an implementation of the in-depth path with numbering first summit:

```
def dfs (V, E):
    # build neighboring peaks
    neighbors = {v: [] for v in V}
    for (u, v) in E:
        neighbors [u] .append (v)
        neighbors [v] .append (u)
    prenum = {v: 0 for v in V}
    i = 0
    # launch the deep excavation on each summit not visit
    for v in V:
        if prenum [v] == 0:
            i = _dfs (v, i, neighbors, prenum)
    return prenum

def _dfs (v, i, neighbors, prenum):
    i + = 1
    first [v] = i
    for w in neighbors [v]:
        if prenum [w] == 0:
```

```
            i = _dfs (w, i, neighbors, prenum)
    return i
```

The deep route gives the order of visit first summit: 1,2,3,6,5,4,7,8.
Graphically, we get the following subtending tree:

Recall that a vertex v is a point of articulation of a non-directed connected graph if the graph obtained by removing v and its incident edges is not connected.

Having done a thorough course on a graph G, we can find his points articulate from the order prenum and the tree underlying T. For each met v, we calculate a new highest value [v]. In order to calculate this value we do a post-order course of T and calculate highest [v] as the minimum among

(a) prenum [v],

(b) prenum [w] for any edge (v, w) ∈ E (G) such that (v, w) / ∈ E (T),

(c) highest [w] for any child w of v.

Let v be a vertex, then v is a point of articulation of G if and only if

- v is the root of T and has at least 2 children, or

- v is not the root of T and has a child w such that highest [w] ≥ prenum [v].

The points of articulation of the graph are therefore the vertices 1 and 4. Here, graphically, the underlying tree after the calculation of highest.

4

Question: Topological sorting of the vertices of the next oriented and acyclic graph.

Solution: All you have to do is go through the graph and number
each vertex at the very end of its exploration, then reverse the order obtained.

The postnum order obtained after the depth course is: 24, 8, 12, 4, 6, 2, 3, 1.

Reverse this order sorts vertices topologically: 1, 3, 2, 6, 4, 12, 8, 24.