Technical Specifications

Python version

Important

Python 3.x (no 2.7...) must be used.

Files containing the lexicon

The input lexicons are simple not empty text files with one lowercase word on each line (be aware of "empty lines" at the end) not necessarily in alphabetic order.

Allowed types

Important

No other structured types than `graph.Graph`, `queue.Queue`, `list`, `tuple` and `str` can be used (no dictionnaries nor sets...) You cannot add your own Class definitions...

`graph.py` and `queue.py` are imported from `algo_py` (you cannot import anything else).

Graphs

We use the adjacency implementation of graphs (class `Graph`) as seen in tutorial.

In the graph built with *k*-length words from a lexicon:

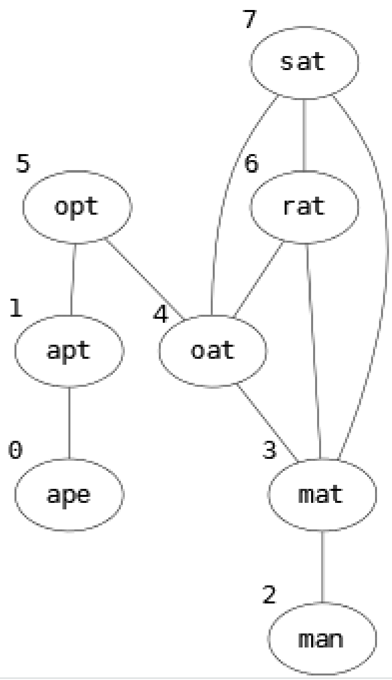
- Each word (with *k* letters) is a vertex (the word is its "label")
- two words are connected if they differ from only one letter (at the same place)
- thus, the graph is simple: no multiple edges, no loops.

There is no empty graphs: each graph have at least 3 vertices and 2 edges.

Labels in `graph.Graph`

Reminder

Our graphs have `labels`. For instance if `G3_ex` is the graph built with words with 3 letters from the file `lexicons/lex_ex.txt`:




```
>>> G3_ex.labels
['ape', 'apt', 'man', 'mat', 'oat', 'opt', 'rat', 'sat']
>>> G3_ex.labels[0]    # gives the label of the vertex 0
"ape"
>>> G3_ex.labels.index("man")    # gives the vertex with label "man"
2
```

Allowed functions

- All methods and functions imported from `algo_py/graph.py` except `todot` and `display`
- All methods imported from `algo_py/queue.py`
- **Classical functions/methods:**
  - `range`
  - "little" functions on int: `min`, `max`, `abs`...
- All functions/methods on `list`
- **All functions/method on `str`, (useful?) example:**
  - `S.strip() -> str`: Return a copy of the string `S` without the last `'\n'` character
- **Any function/method on files, (useful?) examples:**
  - `open(file, mode) -> file object`: `mode = 'r'` open for reading (default)
  - `f.close()`
  - `f.readlines() -> str list`: Read and return the list of all lines remaining in the current file.

Note

Additional functions can be added as long as they begin with `__` and are documented.

Python sources

- All words in lexicons (and thus in graphs) are lowercase words. Each word can only appear once.
- Words passed as parameters have always the correct length for the graph.
- If words are not in the graph, there is no solution...
- No empty lexicons, no empty graphs, no empty words will be used in tests.
- **The graphs used here are both built from `lexicons/lex_some.txt`**
  - `G3` is the graph of 3-length words
  - `G4` is the graph of 4-length words

`doublets.buildgraph`(*filename, k*)

Build a graph with words of length k from a lexicon

Parameters:
- **filename** (*str*) -- the name (and the path) of the file that contains the lexique (one word per line)
- **k** (*int*) -- number of letters of words to build the graph

Returns: the graph where words are vertices and edges are between words that differs from only one letter

Return type: graph.Graph

Examples: The two graphs used in all examples here

```
>>> G3 = buildgraph("lexicons/lex_some.txt", 3)
>>> G4 = buildgraph("lexicons/lex_some.txt", 4)
```

`doublets.mostconnected`(*G*)

The list of words that are directly linked to the most other words in G

Parameters: **G** (*graph.Graph*) -- the graph

Return type: <str> list

Examples: order in the list does not matter

```
>>> mostconnected(G4)
['ford', 'fork']
>>> mostconnected(G3)
['oat', 'sat']
```

`doublets.ischain`(*G, L*)

Test if L is a valid elementary *chain* **in the graph G**

Parameters:

- **G** (*graph.Graph*) -- the graph
- **L** (*<str> list*) -- not empty list that contains words of the supposed chain

Return type: bool

Examples:

```
>>> ischain(G3, ['ape', 'apt', 'opt', 'oat', 'mat', 'man'])
True
>>> ischain(G3, ['man', 'mat', 'sat', 'sit', 'pit', 'pig'])
False
>>> ischain(G3, ['ape', 'apt', 'opt', 'oat', 'mat', 'rat', 'oat',
'mat', 'man'])
False
```

`doublets.alldoublets`(*G, start*)

All words that can form a *doublet* with the start word in the lexicon in G

Parameters:

- **G** (*graph.Graph*) -- the graph
- **start** (*str*) -- the word we search doublets with

Return type: <str> list

Example: order in the list does not matter

```
>>> alldoublets(G3, "pen")
['eel', 'een', 'ell', 'ilk', 'ill', 'ink', 'pie', 'pig', 'pin',
'pit']
```

`doublets.nosolution`(*G*)

Find a *doublet* without solution in G

- Return a pair of words (x, y), both in G, such that there is no solution to the doublet (x, y)
- If no pair is found, return (None, None).

Parameters: **G** (*graph.Graph*) -- the graph

Return type: tuple (str, str) or (NoneType, NoneType)

Examples: there may be other solutions

```
>>> nosolution(G3)
('ape', 'eel')
>>> nosolution(G4)
(None, None)
```

`doublets.ladder`(*G, start, end*)

Find a *ladder* to the *doublet* (start, end) in G with start != end

Parameters:

- **G** (*graph.Graph*) -- the graph
- **start** (*str*) -- the first word of the doublet
- **end** (*str*) -- the second word of the doublet

Return type: <str> list

Examples: there may be other solutions

```
>>> ladder(G3, "ape", "man")
['ape', 'apt', 'opt', 'oat', 'mat', 'man']
>>> ladder(G3, "man", "pig")
[]
>>> ladder(G4, "work", "food")
['work', 'fork', 'ford', 'food']
```

`doublets.mostdifficult`(*G*)

Find in G one of the most difficult *doublets* (that has the longest *ladder*)

Parameters: **G** (*graph.Graph*) -- the graph

Return type: tuple (str, str, int) = (start, end, length) -> (start, end) is the doublet, length is the length of the ladder between them

Example: there may be other solutions

```
>>> mostdifficult(G3)
('one', 'tea', 10)   # ('ape', 'one', 10) is another solution
>>> mostdifficult(G4)
('tree', 'five', 13)
```