On-line exam in Advanced Programming in Python (DAT515)

Chalmers University of Technology
10 January 2022, 14:00 - 18:00, on-line
Examiner: Aarne Ranta aarne@chalmers.se

Do not submit this exam if you have taken the campus exam.

This exam aims to test the most important skills learned from the course. The main skills are

- to read a specification of a possibly unfamiliar task and write a program that solves it,
- to find and use relevant libraries in the solution,
- to read data from text files and convert it to adequate data structures,
- to build and visualize graphs (the running example in the course labs).

You are advised to take the following steps:

- Read the Introduction and all questions first, before writing any code.
- Maintain a conditionally executable main function (specified in Question 3), which you extend after each development step, so that you can test your program continuously.
- Run `python3 exam.py` frequently to test your progress.
- Remember that running this command from the command line is also the first thing we will do when grading your exam.
- Submit your file `exam.py` via Canvas.

You will need 15 points out of 30 in questions 1-3 of this exam to get accepted with the grade that your lab work allows. If you have done extra labs (colouring or clustering) you will also need to get half of the points for the corresponding extra questions. If you have not done extra labs, your answers to those questions will not be graded. The final result of this exam will be reported as 3, 4, 5, or rejected. As specified in the course plan, you will get

- grade 5 if you have at least 50 points from the labs, at least half of the points of the bonus questions corresponding to your labs, and at least 15 points from questions 1-3.
- grade 4 if you have at least 40 points from the labs, at least half of the points of the bonus questions corresponding to your labs, and at least 15 points from questions 1-3.
- grade 3 if you have at least 30 points from the labs and at least 15 points from questions 1-3.
- grade U otherwise

If your lab grading has not been completed yet, your final grade will be determined when the lab grading is complete. Hence you don't need to worry about possibly missing lab points: they will be adjusted later. In particular, since extra lab grading may be delayed, you should do the bonus questions here even if you don't know the outcome of the grading yet.

Introduction: general description of the task

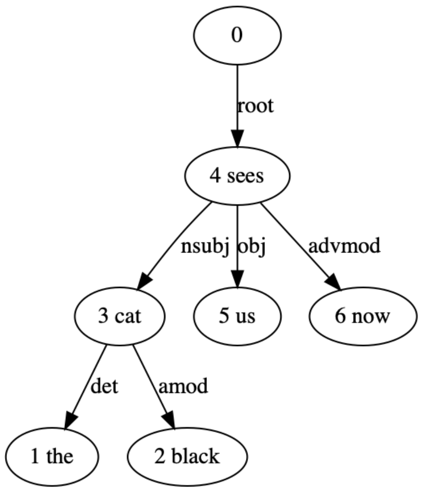
Dependency trees are a format of syntactic analysis used for natural language. A tree is a directed graph that consists of

- vertices, each representing a word and its position in a sentence;
- edges, stating that one word is a **dependent** and another word is its **head**, via a specific **relation;**
- a **root**, which is a vertex with no head and with no word that it represents.

Typical dependent-head relations are

- being the subject or object of a verb (nsubj, obj)
- being an adjective that modifies a noun (amod)

The following tree is a simple example:



It represents the sentence *the black cat sees us now***.** The sentence can be read from the tree by selecting the words in the order in which the words are numbered: 1,2,3,4,5,6.

The standard textual representation of dependency trees looks as follows:

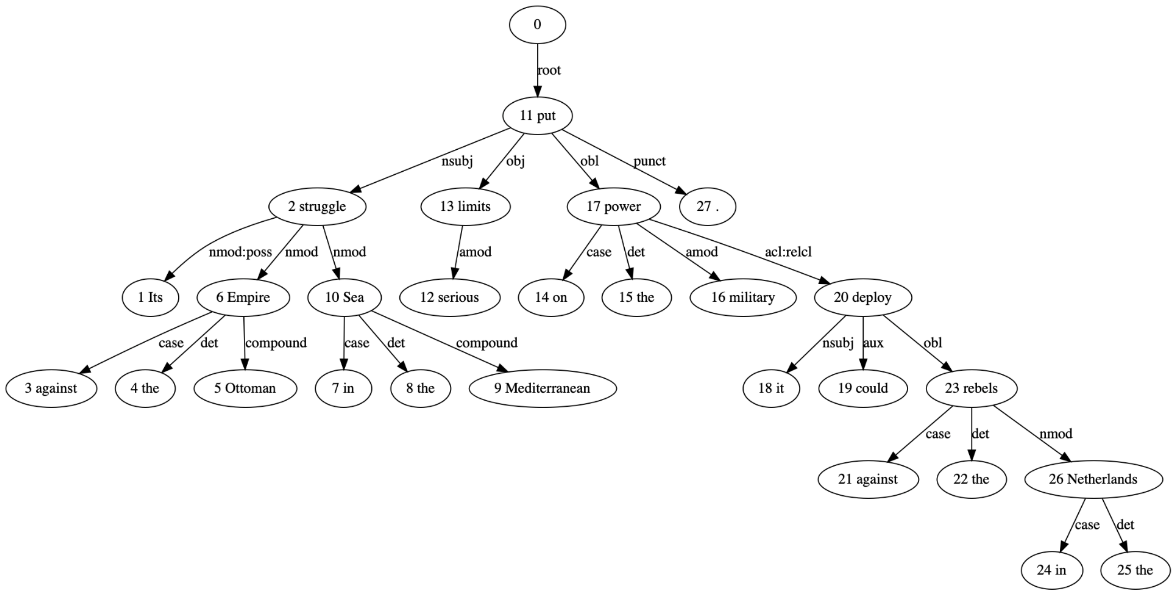
```
# sent_id = gfud1000001
# text = the black cat sees us now
1       the     the     DET     Det     _       3       det     _       _
2       black   black   ADJ     A       _       3       amod    _       _
3       cat     cat     NOUN    N       Number=Sing     4       nsubj   _       _
4       sees    see     VERB    V2      Number=Sing|Person=3|Tense=Pres 0       root    _       _
5       us      we      PRON    Pron    Case=Acc|PronType=Prs   4       obj     _       _
6       now     now     ADV     Adv     _       4       advmod  _       _
```

The sentences are stored as paragraphs in files, each separated by an empty line. The paragraph starts with some metadata (lines prefixed with #) after which the numbered words are listed as lines with tab-separated values. Each column (tab-separated position) gives a specific kind of information. We will be interested in just a few of them:

- column 0: position in sentence
- column 1: the word
- column 3: its part of speech ("word class"), such as NOUN, ADJ, VERB
- column 6: the position of the head of the word
- column 7: the relation between the word and its head

(those with interest in linguistics can figure out the meanings of at least some of the other columns). A file that contains a set of such sentences is called a **treebank**. Treebanks are used as data for supervised machine learning of natural language **parsers**: from a few thousand trees, a parser can be learned that can analyse unseen sentences into the same format with a reasonable accuracy.

In this exam, we will build a few functions for analysing and visualizing treebank files. We will work against a standard treebank, the file `en_pud-ud-test.conllu`. Here is an example tree from that file, as visualized by the function that your task is to define:



The corresponding paragraph in the file has the identifier w01069094.

You will not need to understand the meaning of the relations or the parts of speech, but just the graph structure defined by the positions of the words. But if you want (maybe after the exam itself) a more thorough understanding of dependency trees and parsing, you can take a look at the website https://universaldependencies.org/, where our example file has been obtained from.

Question 0: Statement and Information

In the beginning of your `exam.py` file, include the following comments:

```
# I hereby confirm that I do not communicate with other people than the
teacher of the course during the exam.

# I am aware that cheating in the exam may lead to disciplinary measures.
```

After them, include comments that list

- your name and lab group number (if you have one)
- the URL of your Git repository for the labs
- the extra labs that you have done

Question 1: Analysing treebank files and computing simple statistics (12 p)

Your task is to read a treebank file, in the format specified above, and return a frequency table of parts of speech: how many times does each part of speech appear in the file? The result should be sorted in descending order of frequency and printed line by line. The output will look as follows:

```
PUNCT 2451
VERB 2150
ADJ 1540
PRON 1021
```

We have here left out most of the parts of speech and their frequencies so that you can figure them out yourself.

We recommend you to structure your code into the following functions. One reason to do so is that you can gain points from some of the functions even though some of them might be wrong:

```
# read a file and return a list of non-empty paragraphs
def paragraphs(file):

# convert a paragraph into a structure that represents the words of a sentence
def sentence(para):

# return a structure that says how many times each part of speech occurs in the file
def posfreqs(sentences):
```

Notice that it is your task to decide which structures - lists, tuples, sets, dictionaries, etc - to use as the representations to return. You should think about them as being maximally simple to use in functions that receive them, as well as simple enough to construct. You may want to look at Questions 2 and 3 before deciding, in particular the structure returned by `sentence()`.

You will need to demonstrate your functions with the file

https://www.cse.chalmers.se/~aarne/en_pud-ud-test.conllu

so it is a good idea to download this file at once.

Question 2: visualizing dependency graphs (12 p)

Your second task is to convert the textual representations of dependency graphs into visual images, of the kind shown in the introduction section above. This should be done with the function

```
# convert a sentence to an image that pops up when the function is called
def visualize(sentence):
```

which works on the outputs of your `sentence()` function. You can do this by using a graph visualization library directly, or indirectly by first constructing a graph object. The important things are to

- use the directed graph structure (with arrows from heads to dependents),
- label the edges by relations,
- create distinct nodes for words that appear more than once in the sentences.

To test this function with real data, you will need a working solution of Question 1. However, you can submit a solution even if your Question 1 solution does not work. In that case, work under the assumption that a sentence is a list of lists of strings, where each inner list contains the ten tab-separated values specified in the Introduction.

Question 3: a main function (6p)

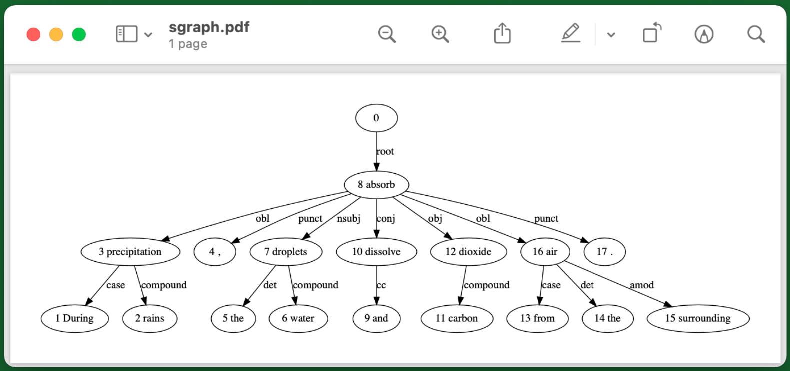
Both for your own testing and the teachers' grading, your file should contain a conditionally executed main function, which does the following things:

1. read the file `en_pud-ud-test.conllu`
2. print the number of paragraphs in that file (to test the `paragraphs()` function)
3. print the sorted frequency table of parts of speech (to test the `posfreqs()` function)
4. select a random number *r* within the range of the number of paragraphs
5. print your representation of sentence number *r* (to test the `sentence()` function)
6. pop up a visualization of your randomly selected sentence (to test the `visualize()` function)

We can promise that the last step is rewarding and fun, and you may just want to run it over and over again and try to read the original sentence by following the position numbers! But we also advise that you add this function to your code as the first thing to do, and test it over and over again when adding functionalities. You can get points for this function even if your answers to Questions 1 and 2 are incomplete or have errors.

This function must be run when your file `exam.py` is executed from the command line, but not when you import the file from the Python shell. When importing your file in the shell, no printing or visualization should happen. What we expect to see is thus the following (the lines starting with # are not a part of the output, but just explanations of what should happen):

```
$ python3 exam.py
1000  # the number of paragraphs (should be 1000 for our test file)
# ... the full frequency table ...
VERB 2150
PRON 1021
# ... should contain more entries
# ... your representation of the random tree ...
# ... a window pops up visualizing the tree ...
```



Question 4 (for those who have submitted the graph colouring lab, 10p)

In the Introduction, two dependency trees were shown. As they are graphs, they can be coloured with your algorithm. Notice that, even though we have shown them as directed graphs, you can treat them as undirected ones if that is easier for you.

Start with the smaller tree, the one for *the black cat sees us*. Try to colour it with two colours, red and green, and show a possible order in which the simplify-select algorithm proceeds, described in the following format (the same as in the lab specification):

1. remove 4
2. …
3. add 2 green
4. …

You can do this by hand, or of course also use your own lab function.

Then turn attention to the bigger tree in the Introduction (*Its struggle against the Ottoman empire…*). How many colours are needed to colour this graph? Try this by hand or use your own lab function. It is enough to give the number of colours as your answer.

Finally, can you say anything in general about graphs that are trees? Can they always be coloured by some small number of colours? Give a proof or a convincing argument.

The answer to this question should be given as a multiline string comment in your exam.py file. You should not include any code that requires importing your lab solution module.

(There are no extra questions for those who have done the second part of graph colouring.)

Question 5 (for those that have submitted the clustering lab, 10 p)

In the Introduction, two dependency trees were shown. As they are graphs, they can be analysed by graph algorithms. Notice that, even though we have shown them as directed graphs, you can treat them as undirected ones if that is easier for you.

In particular, *k*-spanning tree can be used. Since the graphs are already in the tree form, you can skip the step of forming the smallest spanning tree. You should just show what happens when dividing the tree into *k* clusters based on the distance of nodes.

The distance of nodes is defined as the difference of their positions in the sentence. Thus the distance between *sees* and *now* in the smaller tree is 6 - 4 = 2.

Start with the smaller graph, the one for *the black cat sees us now*. Show the clusters that are formed with the *k*-spanning tree method when *k* = 4. You can show them by just listing the sequences of words (e.g. *black cat*) in the resulting clusters.

Then turn attention to the bigger tree in the Introduction (*Its struggle against the Ottoman empire…*). Form clusters with the condition that the distance is at most 2. How many clusters do you obtain? List the corresponding sequences of words.

You can solve these problems by hand or use your own lab function. One simple method is to print or draw the trees on paper, mark the distances at each edge manually, and figure out the result from that drawing.

The answer to this question should be given as a multiline string comment in your exam.py file. You should not include any code that requires importing your lab solution module.