AI Large Practical (2016–17)
Assignment 2

Alan Smaill

5 October, 2016

1 The assignment

Assignment 2 is about an implementation of a system for representing and evaluating arguments, where arguments are for or against a particular claim, backed up by some supporting evidence. In this part of the course, you should

- look at some of the AI literature on argumentation systems in general (there are some starting places on the course page);
- look at an initial implementation of such a system in Python, and aim to get an idea of how it works;
- extend the system in the ways mentioned below;
- provide 3 test examples, as described below;
- finally submit your extended version of the system, suitably commented with reasons for your design choices.

You should write programs and comments by yourself. You are not permitted to

- copy code which someone else wrote for submission to this assignment;
- show your own programs to other students.

Outside these restrictions, you are encouraged to have discussions with your colleagues about concepts, techniques and tools. For more information, please consult the School's plagiarism policy.

2 Submission

For this assignment, you are required to submit:

- program source code, ensuring
  - you make your code as readable as possible;
  - provide appropriate docstrings for classes and methods;
  - where appropriate, provide additional comments to help the reader understand the intention behind the code.

You should include running examples, and documentation, such as a README file, which enables a user to run your system on your examples with minimal fuss.

The deadline for Assignment-1 draft submission is 16:00 on Monday 19th October 2015 to receive informal feedback. Final deadline is 16:00 on Thurs 29th October 2015.

Please use the DICE submit command:[1]

```
% submit ailp 2 <zip-file-of-your-project-directory>
```

The deadline for Assignment 2 submission is 16:00 on Friday 11th November 2016.

3 Task Specification

The following GitHub repository contains a Python implementation of the Carneades argumentation system:
`https://github.com/ewan-klein/carneades`
API documentation (using the Sphinx documentation package) can be found at
`http://ewan-klein.github.io/carneades/`
The implementation follows quite closely a Haskell implementation of the Carneades argumentation system, and you may find it useful to consult this:
`http://www.cs.nott.ac.uk/~psxvb/Papers/tfp2012_abstract.pdf`
– this contains a useful worked example.

The recommended method of getting the code from GitHub is to use git's clone command. Here's how it might look on a DICE machine:

[1] % here is for the DICE terminal prompt; yours will probably look different.

```
% git clone https://github.com/ewan-klein/carneades.git
Initialized empty Git repository in .../ewan/carneades/.git/
remote: Counting objects: 407, done.
remote: Compressing objects: 100% (282/282), done.
remote: Total 407 (delta 131), reused 0 (delta 0)
Receiving objects: 100% (407/407), 740.90 KiB | 416 KiB/s, done.
Resolving deltas: 100% (176/176), done.
```

If you want to clone into a different directory, say `myproj`, do this:

```
% git clone https://github.com/ewan-klein/carneades.git myproj
```

However, assuming, that you've cloned into the default directory, `carneades`, you can `cd` into the Python package directory and try running the code. You'll need to use Python 3; the code has been developed with Python 3.4. If you have a peek at the `caes.py` module, youll see that the start of the file contains lines like this:

```
"""
First, lets create some propositions using the :class:PropLiteral
constructor. All propositions are atomic, that is, either positive
or negative literals.
>>> kill = PropLiteral(kill)
>>> kill.polarity
True
>>> intent = PropLiteral(intent)
>>> murder = PropLiteral(murder)
>>> witness1 = PropLiteral(witness1)
>>> unreliable1 = PropLiteral(unreliable1)
>>> witness2 = PropLiteral(witness2)
>>> unreliable2 = PropLiteral(unreliable2)
...
```

This is a long 'docstring'; lines starting with >>> represent interactive Python commands. The doctest module can be used to execute them. This is a way to test the behaviour of the code, and to provide illustrative calls for documentation purposes. Towards the bottom of the bottom of this docstring, you will see how to initialise a Carneades Argument Evaluation Structure (CAES) and create the various components of a CAES. The `caes.py` module had logging set to `DEBUG` level, but you can easily comment this out at the top of the file if you want. The module also does some recursive call tracing so that you can get a better idea of what steps are involved in evaluating an argument in a CAES.

```
% cd carneades/src/caes
% python3.4 caes.py
```

```
DEBUG: Added proposition murder to graph
DEBUG: Added proposition -murder to graph
DEBUG: Added proposition intent to graph
DEBUG: Added proposition kill to graph
DEBUG: Proposition intent is already in graph
DEBUG: Added proposition -intent to graph
DEBUG: Added proposition witness1 to graph
DEBUG: Added proposition unreliable1 to graph
DEBUG: Proposition -intent is already in graph
DEBUG: Proposition intent is already in graph
DEBUG: Added proposition witness2 to graph
DEBUG: Added proposition unreliable2 to graph
Calling applicable([witness1], [unreliable1] => intent)
DEBUG: Checking applicability of arg2...
...
```

The `caes.py` module depends on igraph . A Python 3.4 compatible version of `igraph` has been installed on DICE. However, the plotting capability of `igraph` depends on Python bindings for the Cairo library, and these are not currently available on DICE.

4 The extension

4.1 Implementing a file-reading capability

As mentioned above, the sample code in the main `caes.py` docstring illustrates how to initialise a CAES. The goal of this task is to carry out the same function by reading in a file, rather than issuing separate Python commands.

For example, assume that you have created a text file called `caesfile.txt`, and you have extended `caes.py` with a new class `Reader`. Then we would like a `Reader` instance to have something like a `load()` method which would take a file object as an argument.

```
>>> caes_data = open('caesfile.txt')
>>> reader = Reader(...) # supply any init parameters
>>> reader.load(case_data)
```

In order for the `load()` method to work, you will have to figure out an appropriate input syntax, and combine this with code for reading information expressed with this syntax from a file and converting into the Python data structures provided by `caes.py`.

4.2 Devising a syntax

You should work out an appropriate syntax that allows information about

- propositions (both positive and negative),
- arguments,
- audience assumptions and weights, and
- and proof standards associated with particular propositions

to be stated in a single text file.

Although there will be trade-offs, your priority should be to make life easy for the user who wants to write down the above information; that is, dont make the syntax of the input file awkward just to ease your job of writing the deserialisation code. You should also ensure that your input files allow the user to add comments if desired; that is, you should implement a comment syntax that your reader is able to understand.

4.3 Deserialisation

You will need to write Python deserialisation code that converts text strings into the Carneades data structures proposed by `caes.py`. This will be the core of your `Reader` class— although you are not required to wrap your file-reading functions into a class, it will probably be a good way of keeping things tidy. You are welcome to use existing Python libraries where appropriate, and of course it is good practice not to re-invent the wheel. Your deserialisation code should not only lead to the initialisation of a `CAES` instance but also carry out error checking on the input file, and give useful error messages if the input is ill-formed.

The code `caes.py` is provided to you as a starting point. You may decide that some of the design decisions in that implementation could be improved. You are free to modify the API as long as the `caes.py` runs at least as well as it currently does. You should also include comments that document and justify your design decisions.

4.4 Examples

Finally:

- provide three text files containing at least 5 arguments each;
- choose example arguments that 'make sense' from a legal point of view;
- ensure that these files can be read and processed; and

- provide regression tests which show what the expected output will be. You can use any Python testing framework you like (e.g., doctest, unittest, nose).

5 Assessment

You submitted system will be tested, following your README instructions; the following aspects will be taken into consideration:

- design of the input syntax
- content of test files are they coherent sets of arguments?
- whether the results are reasonable, including error checking on validity of input
- clarity of your code (including appropriate comments and explanations)
- justification of design decisions (in the form of docstrings or other comments)

You will be expected to build on this system for the second part of the assignment.

A rough guide on marking:

- To pass: a running program, able to read in at least some of the required inputs.
- For an A, require results to be OK, and comments, structuring and explanation as requested.

Efficiency is not a primary concern here; but extra credit is available for efficiency and good style.

References