AI Large Practical (2017–18)
Assignment 2

Alan Smaill

4 October, 2017

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.
- 3 example input files.

Your system will be tested by running on your example files in the DICE environment. To support this, you should supply a script or arrange otherwise allow running of your system by a shell call:[1]

```
% runCaes <testfile>
```

This should result in output showing trace of the working of your system, and final judgement on the top-level propsition.

Please use the DICE submit command, and submit the whole python package directly as a directory (i.e. the directory analogous to `carneades-1`):

```
% submit ailp cw2 <your-project-directory>
```

The deadline for Assignment 2 submission is 16:00 on Friday 10th November 2017.

3 Task Specification

See assignment 1 for information on downloading the initial system, and for initial reading to look at for necessary background.

4 The extensions

4.1 A richer syntax for statements

In the given system, the statements involved are propositional, and the only connective is negation.

You are asked to extend the syntax of propositions by allowing:

- simple propositions as before;

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

- unary and binary predicates (i.e. predicates taking 1 or two arguments);
- constants (ie symbols used as names for individual entities, and also quoted versions of propositions).

So a statement is either a simple proposition, or a unary predicate applied to a constant, or a binary predicate applied to two constants, or the negation of any one of the above.

In the given code, any string can be used as a proposition. You should consider whether it is appropriate or useful to distinguish between different classes of strings when designing this extension.

Because propositions are primitives for the argumentation aspects of the system, this extension should not require any changes to other code.

4.2 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. Further, we want the files to be intuitively legible for non-informaticians: they should have information in a form that makes sense to anyone who might want to explore the argumentation patterns involved.

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 use an input syntax consistent with the description below, 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`.

4.3 Syntax

You are required to implement a reader that will deal with files written using the syntax below. This needs to deal with:

- extended 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 are trade-offs, the aim is to make life easy for the user who wants to write down the above information. Here is an incomplete specification of the grammar for input files, in a form of extended BNF notation.

```
(*
Lexical concerns:
The categories here assume that there is white space
between instances of the categories. For the quoted propositions
taken as constants, this means that if p is a proposition, then
' p '  is a constant.  Allowing 'p' as a constant also is fine,
but ' p ' should be accepted.
*)

<prop> ::= <baseProp>
         | <pred> <name>
         | <name> <pred> <name>
         | "not" <prop> ;
<baseProp> ::= ...
<constant> ::= '"'<prop>'"' ;
         | ...  ;
<pred> ::= ...  ;
<weight> ::=
(* weight is rational in range 0.0 to 1.0                *)
<defaultStandard> ::= "Default proof standard: " <standard>;
<standard> ::= ...

<argument>::= "If"  <andPropList> "then" <prop>  ;
<argument>::= "If"  <andPropList> "then" <prop> "unless" <OrProplist> ;

<andPropList> ::= ...
(* either single prop, or "and" separated list of props.  *)
<orPropList> ::=  ...
(* either single prop, or "or" separated list of props.   *)
<argName> ::= ...

<comment> ::= ...
(* line starting with ##  *)

<inputLines> ::= <inputLine>;
```

```
<inputLines> ::= <inputLine> <newline> <inputLines>;

<inputLine> ::= <comment>
              |  <argName> ":" < argument > ":" <weight>
              |  "Assumption:" <prop>
              | "Proof standard:" <prop> ":" <standard>
              | "Main query:" <prop>;
```

4.4 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.

4.5 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.

5 Assessment

You submitted system will be tested, as described above; the following aspects will be taken into consideration:

- appropriate implementation of more expressive syntax for propositions.
- 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.

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