CS551K — Multi-agent Systems

Argumentation

Overview

In this assessment, you will create an advanced framework for reasoning building on argumentation theory.

Learning Outcomes

This assessment has the following learning outcomes.

1. Understand the theory behind abstract argumentation
2. Be able to implement efficient abstract argumentation solvers
3. Be able to perform non-monotonic reasoning with a knowledge base and preferences
4. Demonstrate the ability to implement components of a multi-agent system

Overall Contribution

This assessment will count for 20% of the overall mark for the module.

Hand-in

This assessment is due at 23:59:59 on the 2nd of Feb 2020. Handing in up to 24 hours late will attract a 10% penalty, while handing in up to 7 calendar days late will attract a 25% penalty. Both penalties will be deducted as a percentage of the mark obtained. Work handed in more than a week late will be treated as a "no paper".

Submission should take place via myAberdeen. Please upload a single .ZIP file containing your source code. If you would like to include explanatory notes, please include a PDF titled README.PDF as part of your zip file.

Please adhere to the formatting instructions, as all outputs will be automatically marked. Marks will be deducted if output formats differ.

Plagiarism

Plagiarism is a serious offence, and will not be tolerated. If you are unsure about whether your work counts as plagiarised, please contact the course coordinator before the submission deadline. For further details, please refer to the Code of Practice on Student Discipline (`https://tinyurl.com/y92xgkq6`).

Assessment Tasks

Abstract Argumentation

Download the file `parser_abstract.py` and at the start of your program, add the import

```
from parser_abstract import read_file
```

Calling `read_file(filename)` where `filename` is a string will return a tuple consisting of a set of arguments and a set of defeats (with the first argument assumed to defeat the second). You can, for example, then read a file passed in from the command line using

```
import sys
from parser_abstract import read_file

af=read_file(sys.argv[1])
```

Task 1a

Given such an abstract argument framework, write a program that prints out all of an argument framework's stable extensions. Your program should be called `task1a.py`. I will call it by running the following command

```
python3 task1a.py file.aaf
```

Where `file.aaf` is the name of a file containing the abstract argumentation framework. Your program should provide this output by using the command

```
print(stable)
```

Where `stable` is a python `list` of lists, each representing a single stable extension, and sorted alphabetically. In other words, given the stable extensions $\{\{b, a, c, f\}, \{d, a\}$, your output should be

```
[['a', 'b', 'c', 'f'], ['a', 'd']]
```

You can achieve this by simply calling `print` on your list of lists.

You can expect that inputs to this program will not exceed 20 arguments.

Task 1b

Create a program called `task1b.py`, which I will call by running the command

```
python3 task1b.py file.aaf
```

This program should output a list of arguments (sorted alphabetically) representing the grounded extension. In other words, given the grounded extension $\{b, a, c, f\}$ your output should be

```
['a', 'b', 'c', 'f']
```

You can expect that inputs to this program will range from 10 to 100000 arguments, and your program should take no more than 10 seconds to run.

Structured Argumentation

For this task, you should use the file `parser.py` which depends on `rule.py` and `atom.py`. The file `argument.py` might also be useful. Feel free to modify these files as required for this assessment, you will definitely need to add some extra methods to some of the files. The files `tandem.gr` and `undercut.gr` contain example inputs file.

Task 2a

Given a structured argumentation framework as input, your task here is to print out the *number* of strict and defeasible arguments which can be generated. You should base your answer on the file `task2a.py`, modifying it as needed.

Your program will be called as per the following example

```
python3 task2a.py tandem.gr
```

Task 2b

Given a structured argumentation framework as input, your task here is to print out the number of attacks (*not defeats*) generated. You should base your answer on the file `task2b.py`, modifying it as needed.

Your program will be called as per the following example

```
python3 task2b.py tandem.gr
```

Task 2c

Given a structured argumentation framework as input, together with the preference principles and rebut used, your task here is to print out the number of *defeats* generated. You should base your answer on the file `task2c.py`, modifying it as needed.

Your program will be called as per the following example

```
python3 task2c.py tandem.gr wd true
```

Here, `wd` stands for the weakest link democratic principle, and `true` denotes restricted rebut. See inside the `task2c.py` file for more details.

Task 2d

Given a structured argumentation framework as input, together with the preference principles and rebut used as well as a semantics, your task is to print out the justified conclusions of the extensions. You should base your answer on the file `task2d.py`, modifying it as needed. Note that this prints out every extension ordered alphabetically and by length.

Your program will be called as per the following example

```
python3 task2d.py tandem.gr wd true preferred
```

Here, `wd` stands for the weakest link democratic principle, and `true` denotes restricted rebut. `preferred` means that the preferred extensions should be generated. See inside the `task2d.py` file for more details.

Task 2e

Finally, your task here is to print out the number of extensions in which a specific argument is labelled **in**, **out**, or **undec**. You should base your answer on the file `task2e.py`, modifying it as needed.

Your program will be called as per the following example

```
python3 task2e.py tandem.gr wd true preferred a
```

Here, `wd` stands for the weakest link democratic principle, and `true` denotes restricted rebut. `preferred` means that the preferred extensions should be generated. `a` is the conclusion in question (you may assume that it will always be a positive, i.e., non-negated literal). See inside the `task2e.py` file for more details, as well as `labelling.py` if needed.

Marking Scheme

Task	Marks	Comments
Task 1a	40%	
Task 1b	10%	marks will be given based on the size of input successfully handled.
Task 2a	7.5%	2.5% will be taken off if contraposition isn't handled correctly
Task 2b	7.5%	2.5% will be taken off if contraposition isn't handled correctly
Task 2c	10%	No marks will be allocated if contraposition isn't handled correctly
Task 2d	10%	No marks will be allocated if contraposition isn't handled correctly
Task 2e	15%	No marks will be allocated if contraposition isn't handled correctly

Marks will be deducted if submission instructions are not adhered to, or output isn't displayed as required.

Changelog

24/01/2020	Initial version