Artificial Intelligence – Programming assignment 2

UNIZG FER, academic year 2019/20.

Handed out: 8. April 2020 Due: 26. April 2020 at 23.59.

Refutation cookbook (24 points)

In the second lab assignment we will implement an automated reasoning system based on refutation resolution. The problem that we will deal with throughout the instructions will be an implementation of a system based on refutation resolution algorithm that will assist you during cooking.

Note: despite the fact that we are using a single example throughout the instructions, your implementation **must** work for **all** files attached with this assignment in reasonable time period (max. 2 minutes). Together with the attached files, your implementation will also be tested with an *autograder* using examples which are not published, so make sure that you cover all the edge cases that might introduce issues to your implementation. Together with the lab assignment, you will get access to a sample of test examples, so you can try running them locally. Please read, in detail, the instructions for input and output formatting in the Section "Autograder instructions" as well as the sample outputs in chapter "Output examples", as any errors will not be tolerated.

1. Data loading

In order to apply our reasoning algorithm to various tasks, we will define a standardised input format for the reasoning algorithm. Regardless of the problem type, input data will always be given in two text files: (1) list of clauses and (2) user commands file. Each text file can contain comment lines, which always start with the `#` symbol and should be ignored.

The **List of clauses** file contains a list of clauses that will be used in our reasoning algorithm. Each line of the file contains a single clause in CNF format. Disjunction is defined with symbol `v`, which contains a single whitespace on each side. Literals are represented as sequences of symbols. The symbols in the sequence are limited to lower and upper case alphabet letters, numbers and the underscore symbol. Lower and upper case letters are considered equal (i.e., `a_1 == A_1`). Consequently, all input data can be transformed to lower-case symbols. Negation is defined with ~ symbol and always precedes the literal that it is applied to.

An example line from the clauses file:

```
~a v b
```

This line contains disjunction (`v`) of literals `a` and `b`, with literal `a` being negated (~).

The **user commands** file contains input data used in the second part of the assignment. Each line of the file contains one clause together with a command symbol. The last two symbols in each line will always be a whitespace followed by one of the following command symbols: `?`, `+` or `-`. A more detailed explanation for each command symbol is given in Section 3. Cooking assistant (12 points).

An example line from the user commands file:
`∼a v b ?`

2. Refutation resolution (12 points)

Once we've implemented the data loading part of the assignment, our next task will be to implement refutation resolution algorithm. When applying refutation resolution algorithm to the clauses defined in our input files, the **last** clause in the clauses file is considered the goal clause (the one that we're trying to derive).

In your implementation of refutation resolution algorithm you should use (1) the **set-of-support control strategy** and (2) the **deletion simplification strategy**, which includes removal of redundant and irrelevant clauses.

Additionally, for a given goal clause, your refutation resolution implementation should output whether the goal clause was derived, and if it was, it should also output the **resolution steps** leading to `NIL`. An example of such output for the clauses in `small_example.txt` file:

```
1. a
2. b v ∼a
3. ∼b v c
=============
4. ∼c
=============
5. ∼b (3, 4)
6. ∼a (2, 5)
7. NIL (1, 6)
=============
c is true
```

Along with each clause that was derived in the resolution process, your implementation should output the ordinal number of that clause, as well as the ordinal numbers of the clauses from which it was derived (denoted inside the brackets in the example). These outputs will not be evaluated with the autograder, but they will be checked during oral examination. In the output of your implementation, clauses given as premises should be printed first, then the negated goal clause (or clauses), and afterwards all the derived clauses.

3. Cooking assistant (12 points)

Once we've successfully implemented the refutation resolution algorithm, our next step is to use it to help ourselves choose a recipe depending on the currently available ingredients. If you cook regularly, you are probably familiar with the situation when you are not sure what to cook, which leads to a tedious decision-making process. Furthermore, if you are forgetful, this decision-making process can become more complicated: once you decide on a specific recipe, you still need to check if you have all the necessary ingredients. In order to circumvent all these problems, we will implement a system that will use literals to track which ingredients we currently have, and also maintain a list of recipes that we often prepare.

Our system should be able to load a cookbook (list of clauses) from a text file that contains our initial knowledge base. The cookbook contains a list of ingredients and recipes

written in clause format. In order to be reusable, our system should support the following actions: (1) queries, (2) clause addition and (3) clause deletion.

The system should support two working modes: (1) interactive and (2) test mode. In the **interactive** mode, our system accepts the list of clauses as input, while user commands (queries, clause addition and deletion) are read from the console. In the **test** mode, system accepts paths to (1) the clauses file and (2) the user commands file as inputs. The user commands from the user commands file should be executed sequentially. This test mode will be used with autograder.

A detailed description of all the functionalities that our system **must** support:

1. Check if the given clause is valid (query)
   - The user inputs a clause along with a symbol `?` to mark it as a query
     `>>> ∼c v a ?`
2. Adding knowledge (clause) into existing knowledge base
   - The user inputs a clause along with a symbol `+` indicating that clause should be added to knowledge base
     `>>> a +`
3. Deleting knowledge (clause) from existing knowledge base
   - The user inputs a clause along with a symbol `-` indicating that clause should be deleted from knowledge base, if present
     `>>> a -`
4. Command for exiting: `exit`

Commands that our cooking assistant must support are the `exit` keyword and clauses followed by a single whitespace and command identifier (`?, +, -`). The `exit` command will not be given in the test files used for autograder, as it will only be used in the interactive mode. The addition and deletion commands (`+`, `-`) **should not** print anything when in test mode.

The last symbol of each query given by user (except for `exit`) will always be the command identifier.

A simple usage example of our system:

```
Resolution system constructed with knowledge:
> a
> ∼b v c
> b v ∼a

Please enter your query
>>> c ?

1. a
2. b v ∼a
3. ∼b v c
=============
4. ∼c
=============
```

```
5. ∼b (3, 4)
6. ∼a (2, 5)
7. NIL (1, 6)
=============
c is true

Please enter your query
>>> ∼b v c -
removed ∼b v c

Please enter your query
>>> c ?
c is unknown
```

Bonus points assignments: smart refutation resolution and automated conversion to CNF (+6 points)

Note: Solutions to all bonus points assignments must be uploaded to Ferko in the same archive together with the rest of the lab assignment.

1. Smart refutation resolution (+2 points)

Through using our cooking assistant, we've noticed that it is possible, and even often, that the system outputs a negative answer to the users' query. However, the system never lets the user know why it couldn't answer their question, even if the value of a single (otherwise unknown) literal could change the decision. This imposes problems regarding the quality of the user experience.

One possible way of improving our cooking assistant is enabling it to ask user some questions, if answers to those questions might help the system find the solution (i.e., answer the original user's question). Of course, such questions should not be too difficult. Therefore, we will limit the possible questions to the values of the literals. In case our cooking assistant wasn't able to derive the goal clause from the premises, our upgraded refutation resolution algorithm should find the list of literals, whose value, if the system had them, might help the system derive the original goal clause.

Using this newly upgraded refutation resolution algorithm, we can improve our cooking assistant. If refutation resolution algorithm wasn't able to derive given goal clause, the system should ask the user for the values of some of the literals. User can then respond that the value for the literal in question is either true, false or unknown (`T, F, ?`). If the system obtained a useful information from the user, it should then restart the refutation resolution algorithm and try to prove the goal clause using this new knowledge.

Think about the easiest way for determining the set of literals that you should ask the user for.

A simple example of our smart resolution system with the `cooking_examples/coffee.txt` input file:

```
>>> Please enter your query
>>> heater -
removed heater
```

```
>>> Please enter your query
>>> coffee ?
coffee is unknown
candidate questions: [hot_water, heater]

>>> hot_water?
>>> [Y/N/?]
>>> N

>>> heater?
>>> [Y/N/?]
>>> Y
coffee is true
```

In the test mode, due to simplicity, the cooking assistant should **only** print the question candidates.

2. Automatic CNF conversion (+4 points)

We've been using a very important simplification in this lab assignment so far, where all the logical expressions we've been using were given in CNF. In this bonus assignment you will need to implement automated CNF conversion.

Input logical expressions will be given in a text file and converter will take a path to that file as input (using the argument that previously was clauses file). Implications will be denoted by `>`, equivalence by `=`, and conjunction by `&`. Disjunction will still be noted by `v`. Each operator will have single whitespace on each side. The elements of each logical operator (reminder: logical operators are **binary**) will be separated with brackets from the rest of the logical expression. The brackets will also explicitly define operator priority. The implication will always be applied from left to right (the symbol `<` will not appear in test files).

An example of logical expression:

$(C \vee D) \Rightarrow (\neg A \Leftrightarrow B)$

The same logical expression written in test file:

```
(C v D) > (~A = B)
```

The expected CNF format obtained with automated CNF conversion for this example:

```
((A v B) v ~C) & ((A v B) v ~D) & ((~A v ~B) v ~C) & ((~A v ~B) v ~D)
```

For each logical expression given in the input file, your implementation should output a single line containing the CNF representation of that logical expression. The order of clauses in the CNF output does **not** matter (as long as the output is logically equivalent to the solution). Additional examples for input and output expressions for CNF conversion can be found in `autocnf/cnfconvert_tests.txt` and `autocnf/cnfconvert_expected_outputs.txt`.

Autograder instructions

Uploaded archive structure

The archive that you will upload to Ferko **has to** be named `JMBAG.zip`, while the structure of the unpacked archive **has to** be as in the following example (the following example is for solutions written in Python, while examples for other languages are given in subsequent sections):

```
|JMBAG.zip
|-- lab2py
|----solution.py [!]
|----resolution.py (optional)
|----...
```

Uploaded archives that are not structured in the given format will **not be graded**. Your code must be able to execute with the following arguments from command line:

1. Assignment flag: string
   One of: `[resolution, cooking_test, cooking_interactive, smart_resolution_test, smart_resolution_interactive, autocnf]`
   The first flag is used for evaluating the first assignment, while the second and third flags are used for evaluating of the second assignment and the rest of the flags are used for evaluating the bonus assignments.
2. Path to the list of clauses file
3. Path to the user commands file (optional)
4. `verbose` flag (optional)
   This flag is not active by default. It is used for generating more detailed outputs that your implementation might have (e.g., the order of clauses checked in the refutation resolution algorithm). These outputs are used only during the oral examination.

Please note that the last two arguments are optional, so it **is possible** that the `verbose` flag is the third place argument.

Your code will be executed on linux. This does not affect your code in any way except if you hardcode the paths to files (which in any way, **you should not do**). Your code should **not use any external libraries**. Use the `UTF-8` encoding for all your source code files.

An example of running your code (for Python):

```
>>> python solution.py resolution resolution_examples/ small_example.txt
> output.txt
```

Instructions for Python

The entry point for your code **must** be in the `solution.py` file. You can structure the rest of your code using additional files and folders, or you can leave all of it inside `solution.py` file. Your code will always be executed from the folder of your assignment (`lab2py`).

The directory structure and execution example can be seen at the end of the previous chapter. Your code will be executed with python version `3.7.4`.

Instructions for Java

Along with the lab assignment, we will publish a project template which should be imported in your IDE. The structure inside your archive `JMBAG.zip` is defined in the template and has to be as in the following example:

```
|JMBAG.zip
|--lab2java
|----src
|------main.java.ui
|--------Solution.java [!]
|--------Resolution.java (optional)
|--------...
|----target
|----pom.xml
```

The entry point for your code **must** be in the file `Solution.java`. You can structure the rest of the code using additional files and folders, or you can leave all of it inside the `Solution.java` file. Your code will be compiled using Maven.

An example of running your code with the autograder (from the `lab2java` folder):

```
>>> mvn compile
>>> java -cp target/classes ui.Solution
```

Info regarding the Maven and Java versions:

```
>>> mvn -version
Apache Maven 3.6.1
Maven home: /usr/share/maven
Java version: 13.0.2, vendor: Oracle Corporation, runtime: /opt/jdk-13.0.2
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "5.3.0-45-generic", arch: "amd64", family: "unix"

>>> java -version
java version "13.0.2" 2020-01-14
Java(TM) SE Runtime Environment (build 13.0.2+8)
Java HotSpot(TM) 64-Bit Server VM (build 13.0.2+8, mixed mode, sharing)
```

Instructions for C++

The structure inside your archive `JMBAG.zip` has to be as in the following example:

```
|JMBAG.zip
|--lab2cpp
|----solution.cpp [!]
|----resolution.cpp (optional)
|----resolution.h (optional)
|----Makefile (optional)
|----...
```

If your submitted archive does not contain a `Makefile`, we will use the Makefile template available along with the assignment. **If** you submit a `Makefile` in the archive

(which we don't suggest, unless you really know what you're doing), we expect it to work.

An example of compiling and running your code with the autograder (from the `lab2cpp` folder):

```
>>> make
>>> ./solution resolution resolution_examples/small_example.txt >
    output.txt
```

Info regarding the gcc version:

```
>>> gcc --version
gcc (Ubuntu 9.2.1-9ubuntu2) 9.2.1 20191008
Copyright (C) 2019 Free Software Foundation, Inc.
```

Output examples

Each output example will also contain the running command that was used to produce that output. Running command assumes Python implementation, but the arguments will be the same for other languages as well.

1. Refutation resolution

For refutation resolution, we've prepared a number of simple examples in the files that will be used for testing your implementation. We will demonstrate outputs for three examples, while the rest of them you can check on your own in the folder `resolution_examples`.

A simple example with a correct result

```
>>> python solution.py resolution resolution_examples/small_example.txt
    verbose


1. a
2. b v ∼a
3. ∼b v c
=============
4. ∼c
=============
5. ∼b (3, 4)
6. ∼a (2, 5)
7. NIL (1, 6)
=============
c is true


>>> python solution.py resolution resolution_examples/small_example.txt


c is true
```

A simple example without NIL (output is the same with and without verbose)

```
>>> python solution.py resolution resolution_examples/small_example_3.txt
```

```
c is unknown
```

An example with descriptive literals: making coffee

```
>>> python solution.py resolution resolution_examples/coffee.txt verbose

1. heater
2. coffee_powder
3. water
4. ∼water v hot_water v ∼heater
5. ∼coffee_powder v coffee v ∼hot_water
===============
6. ∼coffee
===============
7. ∼coffee_powder v ∼hot_water (5, 6)
8. ∼coffee_powder v ∼water v ∼heater (4, 7)
9. ∼coffee_powder v ∼heater (3, 8)
10. ∼heater (2, 9)
11. NIL (1, 10)
===============
coffee is true
```

2. Cooking assistant

An example with descriptive literals: making coffee

```
>>> python solution.py cooking_interactive cooking_examples/coffee.txt
    verbose
```

Output from system initialization **(not necessary in your implementation)**:

```
Testing cooking assistant with standard resolution
Constructed with knowledge:
> heater
> coffee v ∼coffee_powder v ∼hot_water
> water
> ∼heater v ∼water v hot_water
> coffee_powder
```

An example of usage for the cooking assistant:

```
>>> Please enter your query
>>> hot_water ?
1. heater
2. water
3. ∼heater v ∼water v hot_water
===============
4. ∼hot_water
===============
5. ∼heater v ∼water (3, 4)
```

```
6. ∼heater (2, 5)
7. NIL (1, 6)
===============
hot_water is true

>>> Please enter your query
>>> coffee ?
1. heater
2. water
3. ∼heater v ∼water v hot_water
4. coffee_powder
5. coffee v ∼coffee_powder v ∼hot_water
===============
6. ∼coffee
===============
7. ∼coffee_powder v ∼hot_water (5, 6)
8. ∼hot_water (4, 7)
9. ∼heater v ∼water (3, 8)
10. ∼heater (2, 9)
11. NIL (1, 10)
===============
coffee is true

>>> Please enter your query
>>> exit
```

An example of the **test** mode:

```
>>> python solution.py cooking_test cooking_examples/coffee.txt
    cooking_examples/coffee_input.txt > cooking_examples/coffee_output.txt
```

3. Smart refutation resolution

An example with descriptive literals: making coffee

```
>>> python solution.py smart_resolution_interactive
    cooking_examples/coffee.txt verbose
```

Output from system initialization **(not necessary in your implementation)**:

```
Testing cooking assistant with standard resolution
Constructed with knowledge:
> water
> heater
> coffee_powder
> ∼heater v ∼water v hot_water
> coffee v ∼coffee_powder v ∼hot_water
```

An example of usage for the cooking assistant with smart resolution: `cooking_examples/coffee.txt`:

```
>>> Please enter your query
```

```
>>> heater -
removed heater

>>> Please enter your query
>>> coffee ?
coffee is unknown
candidate questions: [hot_water, heater]

>>> hot_water?
>>> [Y/N/?]
>>> N

>>> heater?
>>> [Y/N/?]
>>> Y
coffee is true
```

An example of the **test** mode:

```
>>> python solution.py smart_resolution_test cooking_examples/coffee.txt
    smart_resolution/coffee_input.txt > smart_resolution/coffee_output.txt
```

4. Automated CNF conversion

Input and output example pairs are available in the files `autocnf/autocnf_sample_tests.txt` and `autocnf/autocnf_sample_output.txt`.

An example of running a test for autocnf, which should generate the (logically) equivalent output file:

```
>>> python solution.py autocnf autocnf/sample_tests.txt >
    autocnf/sample_output.txt
```