# Artificial Intelligence – Programming assignment 2

UNIZG FER, academic year 2023/24.

Handed out: March 7, 2024 Due: April 11, 2024 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.

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).

The evaluation of your implementation will be conducted using *autograder*. It is expected that your know how to run your solution with *autograder* in front of the Teaching Assistant during the examinaton. "Autograder instructions" were written by taking into account that **we** will run your solution, which we won't do this year. **You** need to make sure that your solution is runnable and can be evaluated with *autograder* in front of the Teaching Assistant. Together with the lab assignment, you will get access to all test samples.

Deadline for solution submission on Moodle **does not include** the last minute in the day, so make sure that you upload the archive with your solution **before** 23:59. Submissions uploaded exactly at 23:59 or after will be considered as late submissions.

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.

Total number of points in this assignment across the subtasks is set to 24. The number of points that you get will be scaled down to 7.5 points.

### 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 (6 points).

An example line from the user commands file:

`~a v b ?`

## 2. Refutation resolution (18 points)

Once we've implemented the data loading part of the assignment, our next task will be to implement refutation resolution algorithm. When running the refutation resolution algorithm, your code should accept two arguments. First argument will be a keyword "`resolution`" which indicates that refutation resolution algorithm should be run, and the second argument will be the path to the file with the list of clauses.

An example of running solution for resolution in Python:

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

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. ~a v b
3. c v ~b
4. ~c
===============
5. ~b (3, 4)
6. ~a (2, 5)
7. NIL (1, 6)
===============
[CONCLUSION]: 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). 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.

When evaluting your solution with autograder, **only** the last line of the output (prefixed with "`[CONCLUSION]:`") will be evaluated, while the rest of the output will not be checked during oral examination. Therefore, your output in the lines before the final line doesn't have to match the format given in the examples in this document, but the output in those lines is must be descriptive enough so that it can be used for tracking the resolution process. This means that for each derived clause you should output which of the available clauses were used for deriving that clause, which is indicated with ordinal numbers of clauses in brackets in the example above. If you do not output the number of clause, you should output the complete clause with its literals in the brackets. Also, your output should contain only those clauses that led to `NIL`, which means that unused, but derived clauses, must not be printed.

If your output does not contain only those clauses that led to `NIL` and information about the parent clauses for each derived clause, it will be considered as not descriptive enough and you will be penalized during the oral examination.

Also, make sure that the output in the final line of your output is printed in the requested format. In the above given example, the goal clause was successfully derived, so the final line of the output contains message: "`[CONCLUSION]: c is true`". For cases when the goal clause is not derived, in the final line of the output you should print the message: "`[CONCLUSION]: clause is unknown`", where `clause` should be replaced with the given goal clause.

### 3. Cooking assistant (6 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.

For starting the cooking assistant, your solution will be given three arguments. First argument will be a keyword "`cooking`", which denotes that cooking assistant system should be started. Second argument will be a path to the file with the list of clauses, while the third argument will be a path to the file with a list of user commands, from which commands should be read and executed sequentially.

An example of running solution for cooking assistant in Python:

```
>>> python solution.py cooking cooking_examples/coffee.txt
    cooking_examples/coffee_input.txt
```

In this example, the list of clauses is loaded from the `cooking_examples/coffee.txt` file, while user commands are loaded from the file `cooking_examples/coffee_input.txt`.

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 -`

Commands that our cooking assistant must support are clauses followed by a single whitespace and command identifier (`?, +, -`). Your system should **always** output answers to user queries (symbol `?`). The addition and deletion commands (`+`, `-`) **do not have to** print anything.

A simple usage example of cooking assistant system is given below. Output corresponds to above-given command for running the solution in Python (list of clauses from `cooking_examples/coffee.txt` and user commands from `cooking_examples/coffee_input.txt`).

```
Constructed with knowledge:
coffee_powder
~heater v ~water v hot_water
~hot_water v coffee v ~coffee_powder
water
heater

User's command: water ?
1. water
2. ~water
===============
3. NIL (1, 2)
===============
[CONCLUSION]: water is true

User's command: hot_water ?
1. water
2. heater
3. ~heater v ~water v hot_water
4. ~hot_water
===============
5. ~heater v ~water (3, 4)
6. ~water (2, 5)
7. NIL (1, 6)
===============
[CONCLUSION]: hot_water is true
```

```
User's command: coffee ?
1. coffee_powder
2. water
3. heater
4. ~heater v ~water v hot_water
5. ~hot_water v coffee v ~coffee_powder
6. ~coffee
===============
7. ~coffee_powder v ~hot_water (5, 6)
8. ~heater v ~water v ~coffee_powder (4, 7)
9. ~water v ~coffee_powder (3, 8)
10. ~coffee_powder (2, 9)
11. NIL (1, 10)
===============
[CONCLUSION]: coffee is true

User's command: heater -
removed heater

User's command: hot_water ?
[CONCLUSION]: hot_water is unknown

User's command: coffee ?
[CONCLUSION]: coffee is unknown

User's command: heater +
Added heater

User's command: coffee ?
1. coffee_powder
2. water
3. heater
4. ~heater v ~water v hot_water
5. ~hot_water v coffee v ~coffee_powder
6. ~coffee
===============
7. ~coffee_powder v ~hot_water (5, 6)
8. ~heater v ~water v ~coffee_powder (4, 7)
9. ~water v ~coffee_powder (3, 8)
10. ~coffee_powder (2, 9)
11. NIL (1, 10)
===============
[CONCLUSION]: coffee is true
```

As with refutation resolution algorithm, when evaluating your solution for cooking assistant with autograder **only** the lines prefixed with "`[CONCLUSION]:`" will be evaluated, that is the output that represents answers to user queries (given with symbol `?`). Format of conclusion after prefix "`[CONCLUSION]:`" should be the same as in refutation resolution

algorithm.

The order of answers to user queries should be the same as the order of user queries given in the file with user commands. The rest of your output can be structured as you wish, since it will not be evaluated with autograder. However, as in the refutation resolution algorithm, it is necessary that the output for user queries contains the steps of the resolution process, which will be checked during oral examination.

## Autograder instructions

The instructions are given below about uploaded archive structure. Detailed instructions on how to run *autograder* using given structure can be found in `autograder.zip` archive in file `README.md`.

### Uploaded archive structure

The archive that you will upload to Moodle **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`
   The first flag is used for evaluating the first assignment (refutation resolution), while the second is used for evaluating the second assignment (cooking assistant).
2. Path to the list of clauses file
3. Path to the user commands file (only if assignment flag is `cooking`)

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
```

### 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 resolution
    resolution_examples/small_example.txt
```

Info regarding the Maven and Java versions:

```
>>> mvn -version
Apache Maven 3.6.3
Maven home: /opt/maven
Java version: 15.0.2, vendor: Oracle Corporation, runtime: /opt/jdk-15.0.2
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "4.15.0-139-generic", arch: "amd64", family: "unix"

>>> java -version
openjdk version "15.0.2" 2021-01-19
OpenJDK Runtime Environment (build 15.0.2+7-27)
OpenJDK 64-Bit Server VM (build 15.0.2+7-27, 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
```

Info regarding the gcc version:

```
>>> gcc --version
gcc (Ubuntu 9.3.0-11ubuntu0~18.04.1) 9.3.0
```

## 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

We will demonstrate outputs for three examples, while the rest of them you can check on your own. Input files can be found in the folder `resolution_examples`.

**A simple example with a correct result**

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


1. a
2. ~a v b
3. c v ~b
4. ~c
===============
5. ~b (3, 4)
6. ~a (2, 5)
7. NIL (1, 6)
===============
[CONCLUSION]: c is true
```

**A simple example without NIL**

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


[CONCLUSION]: c is unknown
```

**An example with descriptive literals: making coffee**

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

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

### 2. Cooking assistant

An example of output for running the cooking assistant system is given in section "3. Cooking assistant". Due to the length of the output, we will not repeat it here.