I Context

We recall that an *abstract argumentation framework* (AF) is a directed graph $F = \langle A, R \rangle$ where the nodes A represent *arguments*, and the edges R represent the *attacks* between arguments. See Figure 1 for a graphical representation of $F = \langle A, R \rangle$ with $A = \{a,b,c,d\}$ and $R = \{(a,b),(b,c),(b,d)\}$.

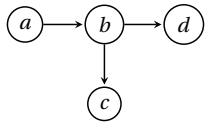

FIGURE 1 – A Simple Example of AF

Given an AF, a set of arguments $E \subseteq A$ is an *admissible set* if and only if

- there are no arguments in E attacking each other, we say that E is *conflict-free*;
- for each argument b which attacks an element of E, there is an element of E which attacks b, we say that E *defends itself* against b.

Formally, E must satisfy these conditions :

- $\forall a, b \in E$, $(a,b) \notin R$ (conflict-freeness) ;
- $\forall a \in E$, $\forall b \in A$, if $(b,a) \in R$ then $\exists c \in E$ such that $(c,b) \in R$ (defense).

For instance, let us consider the AF depicted in Figure 2, the admissible sets are :

FIGURE 2 – Another Example of AF

- $\emptyset$, (the empty set is always an admissible set, because trivially there is no internal contradiction, and there is no need to defend it because it is not attacked),
- $\{A\}$, $\{B\}$, $\{C\}$, $\{D\}$, (each argument is able to defend itself against all the attacks it receives),
- $\{A,C\}$, $\{A,D\}$, $\{B,D\}$, (these are the only sets with two arguments without conflict).

Then, among admissible sets, some of them are called *complete extensions*. These are the admissible sets which contain everything they defend (or, said otherwise, these are the sets which do not defend anything outside of them). So we can say that $E \subseteq A$ is a complete extension if

- E is an admissible set,
- $\forall a \in A$, if E defends a against all its attackers, then $a \in E$.

If we consider again the AF from Figure 1, then the admissible sets are $\emptyset$, $\{a\}$, $\{a,c\}$, $\{a,d\}$, $\{a,c,d\}$. But only $\{a,c,d\}$ is a complete extension, because the other admissible sets do not satisfy the property of "completeness". For instance, $\{a\}$ defends the argument c, but $c \notin \{a\}$.
We can strengthen the constraints over the sets of arguments to define *stable extensions*. A stable extension is a conflict-free set which attacks all the arguments that it does not contain. Formally, $E \subseteq A$ is a stable extension if

- $\forall a, b \in E$, $(a,b) \notin R$ (conflict-freeness) ;
- $\forall a \in A \setminus E$, $\exists b \in E$ such that $(b,a) \in R$ (complement attack).

Notice that all the stable extensions are complete extensions (but the opposite is not true). Also, while there is always at least one complete extension for a given AF, there exist AFs with no stable extensions.

II Instructions

1. Tasks

The goal of the projet is to implement a software which solves these problems :

- Given an AF $F = \langle A, R \rangle$, provide a complete extension of F.
- Given an AF $F = \langle A, R \rangle$ and an argument $a \in A$, determine whether a belongs to some complete extension of F.
- Given an AF $F = \langle A, R \rangle$ and an argument $a \in A$, determine whether a belongs to each complete extension of F.
- Given an AF $F = \langle A, R \rangle$, provide a a stable extension of F.
- Given an AF $F = \langle A, R \rangle$ and an argument $a \in A$, determine whether a belongs to some stable extension of F.
- Given an AF $F = \langle A, R \rangle$ and an argument $a \in A$, determine whether a belongs to each stable extension of F.

In the rest of the document, these problems are denoted SE-CO, DC-CO, DS-CO, SE-ST, DC-ST, DS-ST, where SE-σ means "give **S**ome **E**xtension with respect to the semantics σ", DC-σ means "**D**ecide the **C**redulous acceptability of the argument with respect to σ", and DS-σ means "**D**ecide the **S**keptical acceptability of the argument with respect to σ". The AF will be read from a text file using the following format : each argument is defined in a line of the form

```
arg(name_argument).
```

and each attack is defined in a line of the form

```
att(name_argument_1,name_argument_2).
```

For each line, we assume that there is no white space. All the arguments must be defined before being used in an attack line. Finally, the name of an argument can be any sequence of letters (upper case or lower case), numbers, or the underscore symbole _, except the words `arg` and `att` which are reserved for defining the lines.
An example of a complete file, corresponding to the AF from Figure 1, is given here :

```
arg(a).
arg(b).
arg(c).
arg(d).
att(a,b).
att(b,c).
att(b,d).
```

Your program must be implemented in one of the following languages :

- C,
- C++,
- Java,
- Rust,
- Python.

In the following, we assume that `COMMAND` is either :

- `./executable_file` if you have used C, C++ or Rust,
- `java -jar runnable_jar_file.jar` if you have used Java,

- `python program.py` if you have used Python.

The program will be run with the following command line interface :

- for SE-CO and SE-ST :
  `COMMAND -p SE-XX -f FILE`
  where `XX` is either CO or ST, and `FILE` is the path to the text file describing the AF;
- for DC-CO, DS-CO, DC-ST and DS-ST :
  `COMMAND -p XX-YY -f FILE -a ARG`
  where `XX` is either DC or DS, `YY` is either CO or ST, `FILE` is the path to the text file describing the AF, and `ARG` is the name of the query argument.

The program will then print on the standard output the following result :

- for SE-CO and SE-ST, it prints either a complete extension or a stable extension (if there is one), or `NO` if there is none;
- for DC-CO, DS-CO, DC-ST and DS-ST, it prints `YES` is the query argument is accepted, or `NO` otherwise.

To print an extension, the format is :
`[arg1,arg2,...,argN]`
for representing the set of arguments $\{arg_1, arg_2, \ldots, arg_n\}$. After printing the solution (`YES`, `NO` or a set of arguments), the program prints a new line. For instance, let us assume that the file `af.txt` describes the AF from Figure 1. We describe several command lines and the corresponding outcome of the solver :

```
~$ ./my_solver -p SE-CO -f af.txt
[a,c,d]
~$ ./my_solver -p DC-CO -f af.txt -a a
YES
~$ ./my_solver -p DS-CO -f af.txt -a a
YES
~$ ./my_solver -p DC-CO -f af.txt -a b
NO
```

2. Project delivery

This project must be realized by groups of **two or three students**.
You source code, correctly documented, will be submitted on Moodle before **31/12/2022** (23 :59, Paris time), as a **zip file** containing a directory with the source code, and a makefile or shell script that compiles the code to produce an executable file (in the case of C, C++ and Rust) or a runnable jar file (in the case of Java). Of course, there is no need of such a makefile or shell script if the program is a Python script. You zip file should also contain a pdf file which describes you project. This file must indicate the names of each member of the development team, and it must explain all the algorithms implemented as well as the main data structures. You are free to use algorithms found in the literature on argumentation, in this case you need to explain the algorithm with you own words, and to cite properly the work which originally proposed you algorithm (for instance, in which scientific article this algorithm was proposed).
You zip file will be named with the students names, for instance `Toto_Titi.zip` if the project has been developed by the students Toto and Titi. It contains the pdf file `report_Toto_Titi.pdf` for the project report, and the directory for the source code (and the makefile or shell script, if necessary).