Multiagent systems

CentraleSupélec

Argumentation-based Negotiation for choosing a car engine !

E. Hermellin and W. Ouerdane

16th February 2020

1 The story ...

Imagine that a car manufacturer wants to launch on the market a new car. For this, a crucial choice is the one of the engine that should meet some technical requirements but at the same time be attractive for the customers (economic, robust, ecological, etc.). Several types of engines exist and thus provide a large offers of cars models: essence or diesel Internal Combustion Engine (ICE), compressed natural GAS (CNG), electric battery (EB), fuel Cell (FC), etc. The company decides to take into account different criteria to evaluate them: Consumption, environmental impact (CO2, clean fuel, NO_X[1]...), cost, durability, weight, targeted maximum speed, etc.

The practical sessions in this Multi-Agent System Course will be devoted to the programming of a negotiation & argumentation simulation. Agents representing human engineering will need to negotiate with each other to make a common decision regarding the choice of the best engine. The negotiation comes when the agents have different preferences on the criteria and the argumentation will be used to help them to decide which item to select. Moreover, the arguments supporting the best option will help to build the justification that the engineering should provide to their manager at the end.

> **Notes**. Please note that to implement a MAS correctly and soundly we should do it with the JAVA language programming and use the most popular platform in MAS community namely JADE. As it was not sure the background of each of you regarding JAVA, and as our ultimate goals are to help you to be familiar with the multi agent domain, we choose in this course to use Python and the osBrain platform.

2 Creating a new python project

On your computer, create a new folder who will host this python project. At the root of this new folder, create a file called `run.py`. This will be the main file of your python project. Fill this file with the osBrain *Hello World!* encountered in the previous lab. Run the python script.

```
from osbrain import run_agent
from osbrain import run_nameserver

if __name__ == '__main__':
    ns = run_nameserver()
    agent = run_agent('Example')
    agent.log_info('Hello world!')
    ns.shutdown()
```

You'are now ready to start building the negotiation & argumentation simulation.

[1] Oxyde d'azote

3 Preferences

Each engine is characterized by *values* on a given set of *criteria*. Each agent can attribute different values for the same engine on the same criteria (for example, one agent can rate the engine "Plug-in Hybride[2]" as "low" on the criterion "CO2 emission" while another considers that it is "null' because it assumes that it will be used most of the time under the full (100%) electrical mode). In addition, each agent has a partial or total order on the criteria themselves. This defines the agents' *preferences*.

> **Notes**. In this work, we will not discuss the rating and how the values are obtained, it can be computed by taking into account different sources, different statical analysis, etc.

Exercise 1

1. Create a new python package called `preferences` (be careful to also add the `__init__.py` file in the package).
2. Implement the `Item` class to encode the items (engines). The items should have:
   - A name represented by a `String` value;
   - A descrcription represented by a `String` value;
   - (You can expand this class according to your imagination).

```
class Item:
    """Item class.
    This class implements the objects about which the argumentation will be
    conducted.

    attr:
        name: the name of the item
        description: the description of the item
     """

    def __init__(self, name, description):
        """Creates a new Item.
        """
        self.__name = name
        self.__description = description
```

3. Implement the `Value` and `CriterionName` enumeration classes to encode the values and the criteria names. Feel free to invent whatever criterion you think is relevant!
4. Implement the `CriterionValue` class which associates an `Item` with a `CriterionName` and a `Value`.
5. Implement the `Preferences` class whose instances represent the preferences of participants. The preferences consists of:
   - A list of criteria ordered (from most important to least important);
   - A list of value about each item on each criterion.

   One participant will be associated with a single instance of this class.
6. Write in the `Preferences` class an aggregation function that computes the score of a given object based on its values and the corresponding preferences. The aggregation function will be a simple Ordered Weight Average with arbitrary weights. You may also choose to implement another aggregation function.

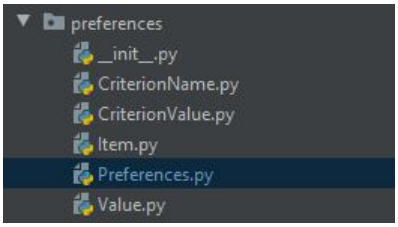
[2] `https://en.wikipedia.org/wiki/Plug-in_hybrid`

7. Test your `Preferences` class: Create some items and check that your methods work.

```
"""Testing the Preferences class.
(you can use unit test structure).
"""
agent_pref = Preferences()
agent_pref.set_criterion_name_list([CriterionName.PRODUCTION_COST, CriterionName
    .ENVIRONMENT_IMPACT, CriterionName.CONSUMPTION, CriterionName.DURABILITY,
    CriterionName.NOISE])

diesel_engine = Item("Diesel Engine", "A super cool diesel engine")
agent_pref.add_criterion_value(CriterionValue(diesel_engine, CriterionName.
    PRODUCTION_COST, Value.VERY_GOOD))
agent_pref.add_criterion_value(CriterionValue(diesel_engine, CriterionName.
    CONSUMPTION, Value.GOOD))
agent_pref.add_criterion_value(CriterionValue(diesel_engine, CriterionName.
    DURABILITY, Value.VERY_GOOD))
agent_pref.add_criterion_value(CriterionValue(diesel_engine, CriterionName.
    ENVIRONMENT_IMPACT, Value.VERY_BAD))
agent_pref.add_criterion_value(CriterionValue(diesel_engine, CriterionName.NOISE
    , Value.VERY_BAD))

electric_engine = Item("Electric Engine", "A very quiet engine")
agent_pref.add_criterion_value(CriterionValue(electric_engine, CriterionName.
    PRODUCTION_COST, Value.BAD))
agent_pref.add_criterion_value(CriterionValue(electric_engine, CriterionName.
    CONSUMPTION, Value.VERY_BAD))
agent_pref.add_criterion_value(CriterionValue(electric_engine, CriterionName.
    DURABILITY, Value.GOOD))
agent_pref.add_criterion_value(CriterionValue(electric_engine, CriterionName.
    ENVIRONMENT_IMPACT, Value.VERY_GOOD))
agent_pref.add_criterion_value(CriterionValue(electric_engine, CriterionName.
    NOISE, Value.VERY_GOOD))

print(diesel_engine.get_value(agent_pref, CriterionName.PRODUCTION_COST))
print(agent_pref.is_preferred_criterion(CriterionName.CONSUMPTION, CriterionName
    .NOISE))
print('Electric Engine > Diesel Engine : {}'.format(agent_pref.is_preferred_item
    (electric_engine, diesel_engine)))
print('Diesel Engine > Electric Engine : {}'.format(agent_pref.is_preferred_item
    (diesel_engine, electric_engine)))
print('Electric Engine (for agent 1) = {}'.format(electric_engine.get_score(
    agent_pref)))
print('Diesel Engine (for agent 1) = {}'.format(diesel_engine.get_score(
    agent_pref)))
print('Most preferred item is : {}'.format(agent_pref.most_preferred([
    diesel_engine, electric_engine]).get_name()))
```

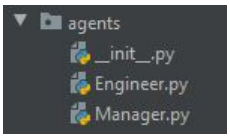
Optionnal Write different methods for your model of preferences: You will need a random method but also a method that receives specific values for testing (either from reading a file or from an internal data structure).

At the end, your preferences package should look like this:



4 Agents

To perform the negotiation & argumentation simulation, we are going to implement two types of agents: Engineer and Manager. To find out how to implement agents with osBrain, you can refer to the previous lab. These two agent classes will be in a new python package called `agents` and will look like this at the end:



4.1 Manager

The first agent to implement is the agent manager that will supervise (or manage) the outcomes of the negotiation. While this can be done by each agent separately, it might be convenient to have one thing at a single spot. Moreover. we recall that in our toy example the final decision (and its validation) is the responsibility of the mananger.

Exercise 2

1. Implement a `Manager` as an osBrain agent. This agent should have:
   - A list of items to discuss.
   - A list of selected items (after negotiation).
   - Print a debug message to indicate agent is initialized.

```python
class Manager(Agent):
    """Manager agent class.
    This class implements the manager agent.

    attr:
        list_items: the list of items to discuss
        selected_items: the list of selected items
    """

    def on_init(self):
        """Initializes the agent.
        """
        self.__list_items = []
        self.__selected_items = []

        """Initialize communication channel here.
        """

    """Other methods here.
    """
```

2. Write the methods and configure the communication channels for the three behaviors of the manager.
   - Answer from request about the items to discuss: Send the list.
   - Request to select an item: Add th item in the dedicated list if negotiation succeeded.
   - Answer from request about the selected items list: Send the list.

4.2 Engineer

The agents will represent the engineerings of the company. For the moment, you will only consider two of such agents. As it was discussed in the course, an argumentation based negotiation agent will have the following architecture. The idea is to be able to implement the different components such that it is possible to run a negotiation between at least two agents.

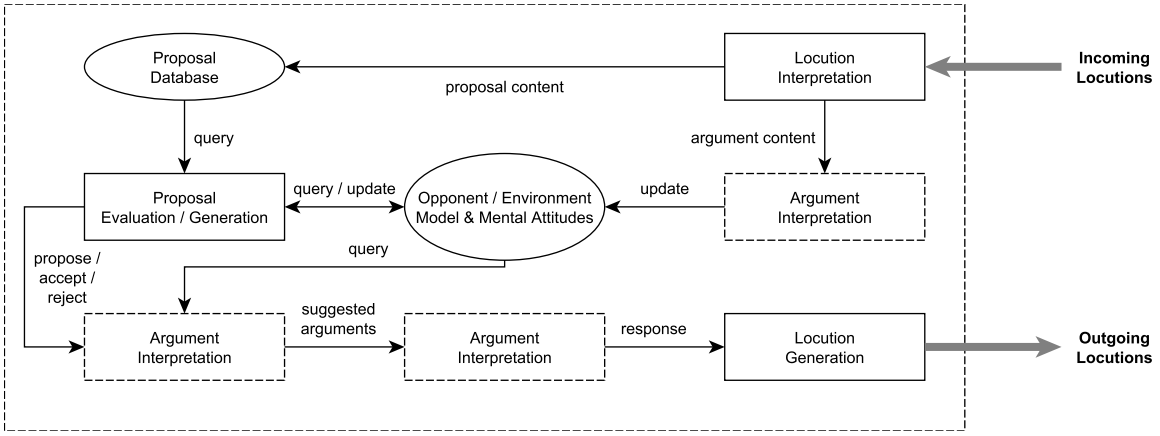


Figure 1: Argumentation based negotiation architecture

Notes. For each block/step, when it is necessary we will highlight the theoretical elements to implement the component. However, for seek of simplicity clarity, and time consideration we will simplify the functioning and the implementation of the components to ease the coding stuff and thus the possibility to go further in the practical work.

Exercise 3

1. Implement a class `Engineering` as an osBrain agent. These agents must have:
   - An instance of `Preferences`, initialized through setup parameters or randomly;
   - Print a debug message to indicate agennt is initialized.

```python
class Engineer(Agent):
    """Engineer agent class.
    This class implements the engineer agent.

    attr:
        preferences: the preferences of the agent
    """

    def on_init(self):
        """Initializes the agent.
        """
        self.log_info("...")
```

2. Write the methods to ask the manager for the list of items, selected items and to select an item.

5 Messages

During the negotiations, participants will exchange messages about the items and their preferences. The communication model (course 3) we are going to implement relies on several performatives. We recall a performative is under the form of $\langle sender, verb(receiver, content)\rangle$, where content is expressed in any knowledge representation language.

Notes. We recall that Agent communication languages (ACLs) define standards for messages exchanged among agents. The most known is the FIPA-ACL (Foundation for Intelligent Physical Agents). Under Jade/JAVA (the MAS platform), we can use directly the ACL message like the folowing;

```
import jade.lang.acl.ACLMessage;
ACLMessage answer = new ACLMessage(ACCEPT);
answer.addReceiver(m.getSender())
answer.setContent(m.getContent());
myAgent.send(answer);
```

However, osBrain does not use the ACL library, so it is not possible to refer to an ACL message directly. Therefore, we will build by ourselves the structure of the message.

5.1 List of performatives

We will use 5 performatives for the negotiation between the two engineerings: PROPOSE, ACCEPT, COMMIT, ASK_WHY and ARGUE. The next subsection explains the role of each performative.

Notes. We will use QUERY_REF and INFORM_REF performative for the interactions between the engineers and the manager.

5.1.1 PROPOSE

This performative is used to propose to select an item. Its content is the name of the item. For example:

```
Agent1 to Agent2: PROPOSE(ICE[3])
```

5.1.2 ACCEPT

This performative is used to accept to select an item. Its content is the item's name. If should always appear after a PROPOSE of the same item, but several other message can exist between the proposal and the acceptance. For example:

```
Agent1 to Agent2: PROPOSE(ICE)
...
Agent2 to Agent1: ACCEPT(Fuel Cell)
```

5.1.3 COMMIT

This performative is used to confirm that an object has to be selected. Its content is an item's name. Both agents must send (and receive) this message before the protocol's initiator can refer about it to the manager agent. Agents must only commit to selecting an item once they are sure to take it, which means:

1. One agent proposed to take it;
2. The other agent (or all other agents in the case of a multiparty negotiation) accepted to take it.

For example:

```
Agent1 to Agent2:  PROPOSE(FC)
Agent2 to Agent1:  ACCEPT(FC)
Agent1 to Agent2:  COMMIT(FC)
Agent2 to Agent1:  COMMIT(FC)
```

Once they have send a COMMIT message, agents consider that the item is no longer in the list of available items.

[3]Internal Combustion Engine

5.1.4 ASK_WHY

This performative is used to ask another agent to propose arguments for a given item. Its content is an item's name. It should be used after a propose. For example:

```
Agent1 to Agent2: PROPOSE(FC)
Agent2 to Agent1: ASK_WHY(FC)
```

This message expects an ARGUE message in answer.

5.1.5 ARGUE

This performative is at the core of the negotiation process. It is used to explain another agent the arguments in favor or against a given item. As will be explained in the theoretical course on negotiation and argumentation, the idea is to communicate a logical inference that defends the agent's position. As a consequence, the content of such a message is of the form:

```
A => B
```

With `A` a logical formula whose terms are called the premises, and `B` a positive or negative literal, which is the conclusion. For example, an agent might want to explain another agent that it favors the item "FC" because it respects the environment, i.e.

```
FC <= ecology impact=high (or CO2 emission= very low)
```

Similarly, an agent might explain another agent that "FC' should not be taken because it is costly

```
not FC <= cost=high
```

Content

In the context of this practical course, we will limit our **premises** to two types of propositions:

- The value of criterion C is X;

  This is a know fact for both agents, but using it in an ARGUE message simply asserts that this information can be used by the sender agent to infer a position in favor or against the item.

  In our communication model, we shall write:

  ```
  C = X
  ```

- Criterion C_1 is more important than criterion C_2;

  This information is local to the sender agent: it does not necessary hold for the receiver. However, using it in an ARGUE message informs the receiver about the sender's preferences. In other words, it gives an argument in favor or against the item.

  In our communication model, we shall write:

  ```
  C1 > C2
  ```

Those two possible types of propositions can be combined in a message. The following paragraphs give some concrete examples.

As for the **conclusion**, it must be the name of an item, preceded by the operator "not" when the inference is against the item.

Example 1

Here is a very simple dialogue in which Agent 2 gives an argument **in favor of** taking the item "FC" based on its value for the criterion "CO2 emission'.

```
Agent1 to Agent2: ASK_WHY(FC)
Agent2 to Agent1: ARGUE (CO2 emission=low => FC)
```

Example 2

Here is a very simple dialogue in which Agent 1 gives an argument **against** taking the item "knife" based on its value for the criterion "food" and its **local** preference for criterion "food" over "transport".

```
Agent1 to Agent2: ASK_WHY(FC)
Agent2 to Agent1: ARGUE (CO2 emission=low => FC)
Agent1 to Agent2: ARGUE(cost=high =>not FC, cost > CO2 emission [4])
```

When several premises are used, they are separated by a comma. This must be interpreted as the conjunction operator (AND).

5.2 Complete example

Here is a sample dialogue with all performatives.

5.2.1 Preferences

Let us consider two agents (Agent1 and Agent2). They have to select only one item between the ICE Diesel (ICED) engine and the Electric (E) one: there is only room left for one of them. The agents consider five different criteria: C_1 : Cost (of production) C_2: Consumption, C_3: durability C_4: Environment impact, C_5: Degree of Noise.

For simplicity, in this example, both agents have exactly the same values for each item.

	$C_1 \downarrow$	$C_2 \downarrow$	$C_3 \uparrow$	$C_4 \downarrow$	$C_5 \downarrow$
ICED	Very Good	Good	Very Good	Very Bad	Very Bad
E	Bad	Very Bad	Good	Very Good	Very Good

The scale for each criterion ranges from Very Bad to Very Good. The majority of criteria are to be minimized $\downarrow$ (the lower the better) except C_3 which is to be maximized $\uparrow$. However, agents have different order of preferences among the criteria themselves (total order in our example):

```
Agent1: Cost > Environment Impact > Consumption >durability  > Noise
Agent2: Environment Impact > Noise > Cost > Consumption >durability
```

5.2.2 Dialogue

The following "fictive" example dialogue could be produced by our agents. Each line is composed of the number of the dialogue turn, the name of the sender agent (the recipient agent is the other one), and the message itself.

```
01: Agent1 - PROPOSE(ICED)
02: Agent2 - ASK_WHY(ICED)
03: Agent1 - ARGUE(ICED <= Cost=Very Good)
04: Agent2 - PROPOSE(E)
05: Agent1 - ASK_WHY(E)
```

[4]This is juste an example and we will not discuss the position of an agent regarding the preference order among the criteria.

```
06: Agent2 - ARGUE(E <= Environment Impact=Very Good)
07: Agent1 - ARGUE(not E <= Cost=Very Bad, Cost>Environment)
08: Agent2 - ARGUE(E <= Noise=Very Good, Noise > Cost)
09: Agent1 - ARGUE(not E <= Consumption=Very Bad)
10: Agent2 - ARGUE(notICED <= Noise=Very Bad, Noise> Cost)
11: Agent1 - ARGUE(ICED <= Durability=Very Good)
12: Agent2 - ARGUE(not ICED <= Environment =Very Bad, Environment >
Durability)
13: Agent2 - ARGUE(E <= Durability=Good)
13: Agent1 - ACCEPT(E)
14: Agent1 - COMMIT(E)
15: Agent2 - COMMIT(E)
```

At this point of the negotiation, the agent that **made the accepted proposal** can send the item `E` to the manager:

```
14: Agent1 to Manager - TAKE(E)
```

5.2.3 Concluding remarks

Note that the inference mechanism is not described in this example. In a future practical session, you will have to implement the reasoning process that computes the necessary arguments in favor or against items and to select which one to use in the negotiation.

It is also possible that two order of preferences conflict in such a way that it is not possible to conclude on a given item. For example, imagine the following exchange of information:

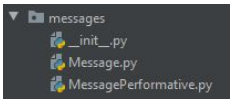
```
Agent1 to Agent2: ARGUE(ICED <= Cost > Environment)
Agent2 to Agent1: ARGUE(not ICED <= Environment > Cost)
```

In such a situation, the agents must infer that they cannot conclude on the proposed item. The initiator should withdraw its proposal and the negotiation should no longer consider it as a candidate for the new engine[5]. You can add a new performative to handle such situations.

5.3 Implementation

Exercice 4:

1. Create a python package called `messages`. At the end, this package will look like:



2. Implement the `MessagePerformative` enumeration class, attributing for each performative a specific number. You shall use specific integer values that do not conflict for each of your newly created performative.

3. Implement the `Message` class. A message should have:

   - An agent sender;
   - An agent receiver;
   - A performative verb;
   - A content.

[5] In a more advanced model, agents could be able to argue about their order of preferences so as to solve such conflicts, instead of dropping the object.

```python
class Message:
    """Message class.
    This class implements the message object which is exchanged between agent
    during communication.

    attr:
        from_agent: the sender of the message
        to_agent: the receiver of the message
        message_performative: the performative of the message
        content: the content of the message
     """
```

4. Adapt the `Engineer` and `Manager` classes to use `Message` during communication.

Exercice 5:

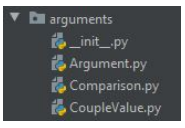
We already test the `Preferences` class. It is time to test communication between one `Manager` and one `Engineer`.

- Modify your `Run.py` to create a simple communication between one `Manager` and one `Engineer`. The created `Engineer` could ask to the manager: to give the items list, to take an item, to give the selected items list.

```python
if __name__ == '__main__':
    """Main programme of the PW3 engine.
    """

    # Init items and preferences

    # System deployment
    ns = run_nameserver()

    # Running agents

    # Setup communication

    # Send messages
```

Exercice 6:

- We recommend that you implement a data structure for the content of ARGUE. Write an `Argument` class that consists of a tuple with:
  - The item name;
  - A boolean value (for positive or negative arguments);
  - A list of couples that can be either $(criterion, value)$ and/or $(criterion, criterion)$.
- You can create a python package called `arguments` to put the `Argument` class inside.



- You can create a `Comparison` and `CoupleValue` classes to implement the list of couples you can find in the `Argument` class.

Using the required getter, setters and `__str__` methods, you will be able to use this data structure both for the decision model and for the communication.

6 Protocols

Agents will follow specific protocols when interacting. We shall use the AUML standard to draw each protocol.

Exercise 7

1. Draw the protocol that correspond to a proposal with an ACCEPT or an ASK_WHY.
2. Implement this protocol in your agent, with a random selection between ACCEPT and ASK_WHY.
3. Test this protocol with random values. No decision is made by the agent at this point.

Exercise 8

1. Draw the protocols that correspond to a request for explanation (ASK_WHY followed by one single ARGUE).
2. Implement this protocol in your agent, with a random selection for the explanation's content. Again, no decision is made at this point.
3. Test this protocol.

7 Decision and argumentation

In order to make decisions about the items to propose, to accept, to attack or to defend, the agents will required several features:

- Find their favorite item in the current list of opened item (given by the `Manager`);
- Decide whether or not to accept a proposal;
- Resolve the conflicts with the exchanged arguments.

In this first session, we will only consider two `Engineer` agents, and one `Manager` agent.

7.1 Finding the favorite item (Proposal Generation/Evaluation)

Notes. Proposal generation is usually made as a result of some utility evaluation or planning process. For instance, the agent may use an underlying planner to generate a set of actions or resources needed to achieve some intention. Agents then request the action or resources they cannot achieve or to obtain on their own from other agents.

In this toy example we have a multiple criteria problem (see Decision Modeling Course) and we choose to use a simple aggregation function to compute the score of the items and to select the one that maximises this score. It is clear from a theoretical point of view and even from a practical one the weighted sum is not the most suited procedure (for instance due to compensation effect). The idea is just to simplify the task but feel free to implement a more sophisticated procedure. Moreover, we assume that we know the ordering of the criteria (and thus a distribution of importance among them), this is also a big assumption as you have seen in the course of Decision Modeling, this kind of information can not be obtained explicitly but by using an adequate preference learning (elicitation) mechanism.

Agents will always propose their favorite item.

Exercise 9

1. Generate two `Engineer` agents and one `Manager` agent with a list of items, with different criteria and values. The list of items is the same for both agents. Each agent however has its own set of preferences, as seen in previous exercises (1, 2 and 3).

2. Using the methods implemented in previous exercises (2 and 3) and the OWA that computes the value of each item, write a method that allows an agent to select its most preferred item in a list. In case of equality, select one randomly.

optionnal Write and implement a protocol in which both `Engineer` agents register to the `Manager` agent as soon as they are ready to make their first proposal. The `Manager` agent waits for both agents to be ready and calls for the first one that registered. This agent will then contact the other one with its proposal.

7.2 Deciding to accept for a proposal

Agents will accept any proposal that corresponds to an item that belongs to the top 10% of its preferred items list.

Exercise 10

1. Write a method that checks whether an item belongs to the 10% most preferred one of the agent.

2. Use this method to decide between ACCEPT and ASK_WHY in the protocol.

3. In case of acceptance, write and implement the protocol that corresponds to the double-COMMIT interaction. Both agents simply send a COMMIT message to their interlocutor as soon as they have the two following informations:

   - One agent made a proposal on the item;
   - The other agent accept the proposal on the item;

4. Implements the TAKE protocol, that is triggered by the agent that proposed the item that was accepted. This agent sends a message to the `Manager` agent.

5. The `Manager` agent and both `Engineerings` agents shall update the list of available items.

7.3 Generating arguments

> **Notes**. Here we are concerned with the block argumentation generation. It consists in generating candidate arguments to present to a dialogue counterpart. These arguments are usually sent in order to entice the counterpart to accept some proposes agreement. In negotiation argument and proposal generation are closely related processes. Different strategies exist to generate arguments but we did not discuss this in the course. Indeed,

In order to answer to an ASK_WHY performative, agents must be able to write the content of the ARGUE message, following the data structure from exercise 6. Several approaches are possible. In this project, we will extract the list of arguments **during the construction of the initial proposal**. For example, in the dialogue from section 5.2.2, Agent 1 proposes to take the item "ICED". The argument, used at turn #3 of the dialogue, is based on the value of the criterion "Cost" for this item. The algorithm extracts the list of criteria on which the item has a "good enough" value. In our example, we can assume that "Very Good" and "Good" are the "good enough" values. This threshold must be a parameter for each criteria. The agent then selects the criterion, in this list, with the highest position in its own preference order.

Exercise 11

1. Write a method that computes the list of possible arguments for defending a proposal.
2. Implement and test a decision procedure that answer with an ARGUE message when a proposal is questioned.

When one agent receives an ARGUE message, three situations are possible:

- If the agent cannot contradict the argument, it shall accept the proposal;
- If it can contradict the argument, it can answer with another ARGUE message;
- It can also answer with a PROPOSE message, which is another manner to attack the argument. In the example from section 5.2.2, turn #6 attacks turn #3.

The agent can contradict the argument if:

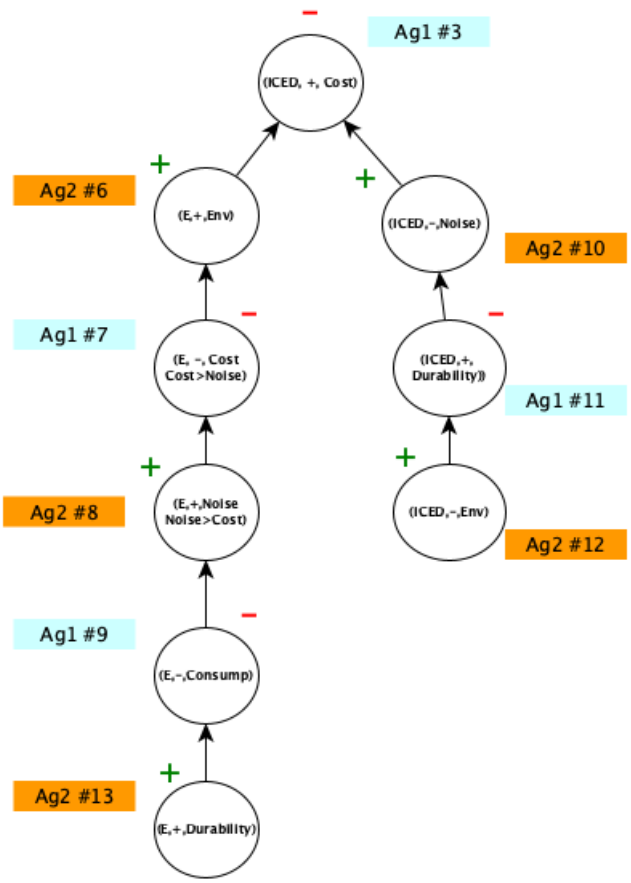
- The criterion is not important for him: This means that some criterion with a "low value" ("Bad" or "Very Bad") is more important in its own preference order. In the example from section 5.2.2, the agent considers that noise is more important than cost. This is used in turn #10 to attack turn #3.

Exercise 12

1. Write a method that decides whether an argument can be contradicted or not;
2. Adapt the behavior of `Engineer` to answer to an ARGUE message based on the possibilities of contradiction;
3. Test your behaviour on a concrete example;
4. You can deal with blocking situations (*e.g.* when two agents have opposite preferences) by terminating the protocol. The item is no longer considered as a possible candidate.

7.4 Evaluating arguments

Exercise 13



8 Report and following steps

8.1 Report

We expect that you write a 5 to 10 pages report about your negotiation model. Give some printouts of negotiations. Compute some statistics on the negotiation outputs depending on the simulation parameters.

You should clearly identify the parameters and the observed values.

8.2 Extensions

If time allows, you should consider the possible extensions to your work:

- Multiple party negotiation. When more than two agents negotiate, you must define a policy for the conversation turn and decide how to make the final decision on a particular item.
- Use a different aggregation rule for finding local the order on the items;
- Change the order of preferences when one agent accepts a proposal, based on the values that were exchanged during the negotiation;
- Consider additional information on the agents that could serve as explanations for the criteria values and that could be used during the negotiation.
- ...

9 Supplementary materials

To carry out the exercises of this practical work, you could need some tools not described previously. Below are classes you may need.

9.1 Argumentation graph

During the negotiation, you may choose to represent the arguments as a graph which you will then go through to resolve this argument. You can find below a generic `Graph` object.

```python
class Graph(object):
    """Graph Class.
    A simple Python graph class, demonstrating the essential
    facts and functionalities of graphs.
    """

    def __init__(self, graph_dict=None):
        """ initializes a graph object
            If no dictionary or None is given,
            an empty dictionary will be used
        """
        if graph_dict is None:
            graph_dict = {}
        self.__graph_dict = graph_dict

    def vertices(self):
        """ returns the vertices of a graph """
        return list(self.__graph_dict.keys())

    def edges(self):
        """ returns the edges of a graph """
        return self.generate_edges()

    def add_node(self, node):
        """ If the node "node" is not in
            self.__graph_dict, a key "node" with an empty
            list as a value is added to the dictionary.
            Otherwise nothing has to be done.
        """
        if node not in self.__graph_dict:
            self.__graph_dict[node] = []

    def add_edge(self, edge):
        """ assumes that edge is of type set, tuple or list;
            between two vertices can be multiple edges!
        """
        edge = set(edge)
        (node1, node2) = tuple(edge)
        if node1 in self.__graph_dict:
            self.__graph_dict[node1].append(node2)
        else:
            self.__graph_dict[node1] = [node2]

    def generate_edges(self):
        """ A method generating the edges of the
            graph "graph". Edges are represented as sets
            with one (a loop back to the node) or two
            vertices
        """
        edges = []
        for node in self.__graph_dict:
            for neighbour in self.__graph_dict[node]:
                if {neighbour, node} not in edges:
                    edges.append({node, neighbour})
```

```
        return edges

    def find_path(self, start_node, end_node, path_found=None):
        """ find a path from start_node to end_node
            in graph """
        if path_found is None:
            path_found = []
        graph_to_search = self.__graph_dict
        path_found = path_found + [start_node]
        if start_node == end_node:
            return path_found
        if start_node not in graph_to_search:
            return None
        for node in graph_to_search[start_node]:
            if node not in path_found:
                extended_path = self.find_path(node, end_node, path_found)
                if extended_path:
                    return extended_path
        return None

    def __str__(self):
        res = "vertices: "
        for k in self.__graph_dict:
            res += str(k) + " "
        res += "\nedges: "
        for edge in self.generate_edges():
            res += str(edge) + " "
        return res

if __name__ == "__main__":
    g = {"a": ["d"],
         "b": ["c"],
         "c": ["b", "c", "d", "e"],
         "d": ["a", "c"],
         "e": ["c"],
         "f": []
         }

    graph = Graph(g)

    print("Vertices of graph:")
    print(graph.vertices())

    print("Edges of graph:")
    print(graph.edges())

    print("Add node:")
    graph.add_node("z")

    print("Vertices of graph:")
    print(graph.vertices())

    print("Add an edge:")
    graph.add_edge({"a", "z"})

    print("Vertices of graph:")
    print(graph.vertices())

    print("Edges of graph:")
    print(graph.edges())

    print('Adding an edge {"x","y"} with new vertices:')
    graph.add_edge({"x", "y"})
    print("Vertices of graph:")
    print(graph.vertices())
    print("Edges of graph:")
```

```
    print(graph.edges())

    print('The path from node "a" to node "b":')
    path = graph.find_path("a", "b")
    print(path)

    print('The path from node "a" to node "f":')
    path = graph.find_path("a", "f")
    print(path)

    print('The path from node "c" to node "c":')
    path = graph.find_path("c", "c")
    print(path)
```