Multiagent systems

CentraleSupélec

Argumentation-based Negotiation for choosing a car engine !

Emmanuel Hermellin, Wassila Ouerdane, Nicolas Sabouret

18.03.2022

1 The story ...

Imagine that a car manufacturer wants to launch a new car on the market. For this, a crucial choice is the one of the engines 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 offer 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. To establish the best offer/choice among a large set of options, they decide to simulate a negotiation process where agents, with different opinions and preferences (even different knowledge and expertise), discuss the issue to end up with the best offer. The simulation will allow the company to simulate several typologies of agent behaviours (expertise, role, preferences, ...) at a lower cost within a reasonable time.

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 joint decision regarding choosing the best engine. The negotiation comes when the agents have different preferences on the criteria, and the argumentation will help them decide which item to select. Moreover, the arguments supporting the best choice will help build the justification supporting it, an essential element for the company to develop its marketing campaign.

Warning. As we are limited in the time and the idea is not to built at the end a software, but to understand the different concepts described in the course, we will take the following assumptions to ease the programming:

- the example illustrates only two agents for the moment !
- The agents share the same set of options (items) and the same set of criteria.
- The negotiation protocol is run only between each pair of agents.
- We will not update or modify the knowledge base of an agent.

2 An illustrative Example

Let consider two agents: A_1 and A_2. 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:

[1] Nitrogen oxyde

- c_1: cost of production (€, lower is better),
- c_2: consumption (L/100 km, lower is better),
- c_3: durability (measured on a qualitative scale ranging from 1 (lowest performance) to 4 (best performance), higher is better),
- c_4: environmental impact (measured on a qualitative scale ranging from 1 (low environmental impact) to 4 (high environmental impact), lower is better),
- c_5: degree of Noise (measured in dB, lower is better)

For illustration, the performances of the two engines, on each criterion, are described in Table 1

	c_1	c_2	c_3	c_4	c_5
ICED	12 330	6.3	3.8	4.8	65
E	17 100	0	3	2.2	48

Table 1: Engines Data

Moreover, each agent has it own evaluation table for the items. This table is constructed by definining threasholds, called *profiles*, that separate scale of values of each criterion between very bad($\star$/■), bad ($\star\star$/■) good ($\star\star\star$/■) and very good ($\star\star\star\star$/■). These *profiles* represent the preferences of the agent. For instance, Table 2 below gives the profiles of A_1 for c_1 and c_3, and the resulting labeling of the two engine according to c_1 and c_3 is depicted in Figure 1.

Profile	c_1	c_3
$p^{1\star}$	17 250	1.5
$p^{2\star}$	15 500	2.5
$p^{3\star}$	12 500	3.5

Table 2: Limiting Profiles for A_1

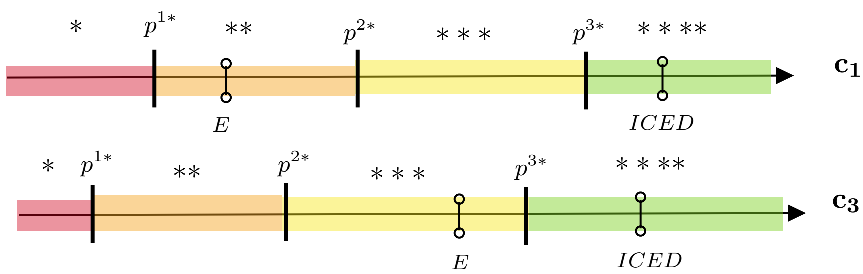


Figure 1: Representation of performances w.r.t. category limits for c_1 and c_3.

Notes. The values given here are abitrary values. We shall not discuss the choice of the criteria, the possible scales, or the rating for each engine. You may think about preference learning and elicitation task in decision theory to obtain these values, but this is out of the scope of this course.

The resulting labeling for both candidate engines on all 5 criteria for agent A_1 is given into Table 3. What matters is this table:

	c_1	c_2	c_3	c_4	c_5
ICED	$\star\star\star\star$	$\star\star\star$	$\star\star\star\star$	$\star$	$\star\star$
E	$\star\star$	$\star$	$\star\star\star$	$\star\star\star\star$	$\star\star\star\star$

Table 3: Performance Table of A_1

Concerning the agent A_2. Table 4 and Table 5 give respectively the limiting profiles and the performance table for agent A_2.

Profile	c_1	c_3
$p^{1^\star}$	20 500	1.8
$p^{2^\star}$	18 000	2.0
$p^{3^\star}$	12 300	3.5

Table 4: Limiting Profiles for A_2

The resulting labeling for both candidate engines on all 5 criteria for agent A_2 is given into Table 5. What matters is this table:

	c_1	c_2	c_3	c_4	c_5
ICED	$\star\star\star$	$\star\star$	$\star\star\star$	$\star$	$\star$
E	$\star\star\star$	$\star\star$	$\star\star$	$\star\star\star\star$	$\star\star\star\star$

Table 5: Performance Table of A_2

Moreover, agents have different order of preferences among the criteria themselves (total order in our example):

- $A_1 : c_1 \succ c_4 \succ c_2 \succ c_3 \succ c_5$
- $A_2 : c_4 \succ c_5 \succ c_1 \succ c_2 \succ c_3$

Warning.
In the rest of the work, we have two ways to generate the performance tables of A_1 and A_2 :

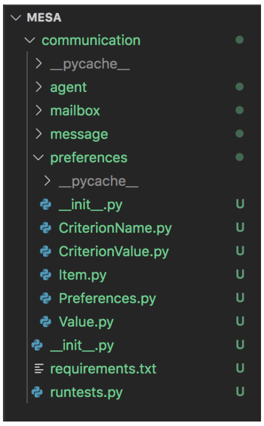
1. Randomly generate the two tables 3 and 5,
2. Randomly generate the profiles for each agent. Then, we construct the performance table by relying on the profiles and the Engines Data.

A basic version of the work is to adopt the first way and then move to the second one to improve the proposed code.

3 The Python project

In the third session of the course, part "Interaction with mesa library", we have used a communication layer in Mesa to handle the direct interactions on the Alice/Bob example. In this practical work we will use this layer and add argumentation and negotiation features. To do so, you can download from EDUNAO the mesa package. Thus you should have the following, such that

- communication: the root folder of the communication layer;
- agent: the folder which will contain the implementation of the communicating agent class;
- mailbox: the folder which will contain the implementation of the mailbox class;
- message: the folder which will contain the implementation of the message and performative class.



Warning.
The provided packages and all the code only help you start. You are free to use the code as it is or to change the structure and the content of the code. Mainly, you are free to modify the functions, names, arguments, and everything you judge necessary to change.

4 What is your goal?

To give an idea of what is the target of this practical work, in what follows is the algorithm 1 of the argumentation-based negotiation that we will try to implement. You can examine the algorithm to get a first idea of the structure and content. A complete understanding of the algorithm is not necessary at the moment, as the different steps in the next sessions will allow you to build it.

Moreover, to have a quick understanding of the algorithm, the following "fictitious" 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)
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)
```

Algorithm 1: Example of a communication protocol

```
initialization;
    • create a set of $n$ agents ;
    • set-up a list of $m$ items $\{o_1, \dots, o_m\}$
    • set-up a list of $k$ criteria $\{c_1, \dots, c_k\}$
foreach agent $i$ in agents do
    define preferences of $i$ (criteria order, a performance table)
end
foreach $(X, Y) \in$ agents do
    $X$: propose ($o_i$);
    $Y$: receivers-message(propose($o_i$));
    if ($o_i \in 10\%$ favorite items of $Y$) then
        if ($o_i = o_j \mid o_j$ top item of $Y$) then
            $Y$: accept($o_i$);
            $X$: commit($o_i$);
            $Y$: commit($o_i$);
        end
        else
            $Y$: propose($o_j$)
        end
    end
    else
        $Y$: ask_why($o_i$)
    end
    $X$: receive-message(ask_why($o_i$));
    if ($X$ has at least one argument pro $o_i$) then
        $X$: argue($o_i$, reasons);
    end
    else
        $X$: propose($o_j$), $j \neq i$
    end
    $Y$: receive-message(argue($o_i$, reasons))
    if ($Y$ has a counter-argument($o_i$) then
        switch (if reasons correspond to ($c_i = x$) ) do
            case ($Y$ has a better alternative $o_j$, $j \neq i$, on $c_i$) do
                $Y$: argue($o_j$, $c_i = y$, $y$ is better than $x$)
            end
            case ($o_i$ has a bad evaluation on $c_i$) do
                $Y$: argue(not $o_i$, $c_i = y$, $y$ is worst than $x$)
            end
            case ($o_i$ has a bad evaluation on $c_j (j \neq i)$ and $c_j$ is an important criterion for $Y$ ) do
                $Y$: argue(not $o_i$, $c_j = y$ with $y$ is worst than $x$, $c_j \succ c_i$)
            end
        end
        switch (if reasons correspond to ($c_i = x$ and $c_i \succ c_j$) ) do
            case ($Y$ has a better alternative $o_j$, $j \neq i$, on $c_i$) do
                $Y$: argue($o_j$, $c_i = y$, $y$ is better than $x$)
            end
            case ($Y$ prefer $c_j$ to $c_i$) do
                $Y$: argue(not $o_i$, $c_j = y$, $c_j \succ c_i$)
            end
        end
    end
    else
        $Y$: accept($o_i$);
        $X$: commit($o_i$);
        $Y$: commit($o_i$);
    end
end
```

5 Agents' Preferences

Warning.
From now de not forgot to make unit tests for each function implemented to check the behaviour of your code.

5.1 The preference package

The preference package includes different classes described in what follows:

- `Item` class: encodes the items (engines). Each item is described by a name represented by a String value and a description represented by a String value;
- `CriterionName` class: implements the possible criterion name (e.g. environment impact, Cost, etc.)
- `CriterionValue` class : associates an Item with a CriterionName and a Value.
- `Value` class: implements the Value class Enumeration containing the possible values (e.g. Bad, Good, etc.)
- `Preferences` class: whose instances represent the preferences of agents. One agent will be associated with a single instance of this class. The preferences consists of: a list of criteria ordered (from most important to least important) and a list of value about each item on each criterion.

Warning. For testing your agents during this session, please consider the preferences of Tables 3 and 5 and the criteria ordering corresponding to A_1 and A_2

5.2 Testing you `Preference` class

1. Using the methods present in the `Preferences` class and the get_score(self, preferences) function that computes the value of each item (see `Item` class), add to your class a method that allows an agent to select its most preferred item in a list. In case of equality, select one randomly. If you think that it is helpful to add another function of your choice, please feel free to do it.

```
def most_preferred(self, item_list):
     """Returns the most preferred item from a list.
     """
    # To be completed
     return best_item
```

2. Add to your class a method that checks whether an item belongs to the 10% most preferred one of the agent.

```
def is_item_among_top_10_percent(self, item, list_items):
    """
    Return whether a given item is among the top 10 percent of the preferred
   items.

    :return: a boolean, True means that the item is among the favourite ones
    """
    # To be completed
```

3. Run the unit tests at the end of the `Preferences` class (Preferences.py) to check wether you code is correct or not. You should get the output of the Figure3.

You can add other units tests not included in the file.

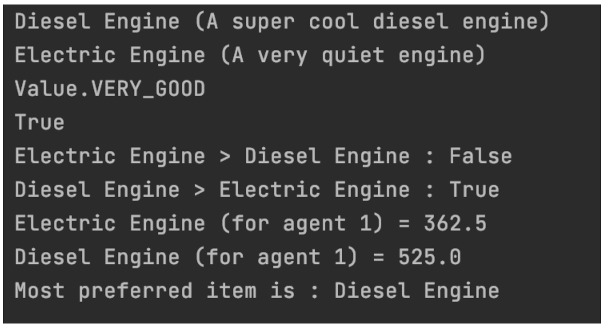


Figure 2: Outputs Preferences Class

6 Agents and Messages

6.1 Performatives and Messages content

During the dialogue (negotiation), we will need some locutions/performatives. we will mainly consider the following:

- PROPOSE: it is used to propose to select an item. Its content is the name of the item.
- ACCEPT: it is used to accept to select an item. Its content is the item's name. It should always appear after a PROPOSE of the same item, but several other message can exist between the proposal and the acceptance.
- COMMIT: it 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 at the end of the interaction.
- ASK_WHY: it 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. This message expects an ARGUE message as answer.
- ARGUE: it is used to explain to the other party in the dialogue the reasons in favor or against a given item.

You should check that these performatives are implemented in MessagePerformative.py, if not make sure to add the missing ones. Again, you can add other perfomatives (for instance, REJECT. You need then to update the protocol to integrate this new locution).

6.2 Agent and Communication

1. Create a new Python file named pw_argumentation.py (as it is illustrated in the following picture) at the root of the mesa folder

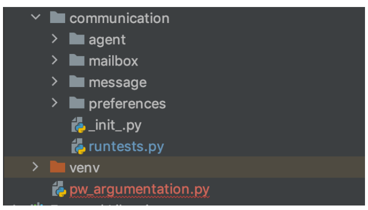


Figure 3: Outputs Preferences Class

This file will contain our class of agents and our model for the argumentation simulation (will be completed as we go through our sessions).

```python
from mesa import Model
from mesa.time import RandomActivation

from communication.agent.CommunicatingAgent import CommunicatingAgent
from communication.message.MessageService import MessageService


 class ArgumentAgent(CommunicatingAgent):
  """ ArgumentAgent which inherit from CommunicatingAgent.
  """
  def __init__(self, unique_id, model, name, preferences):
      super().__init__(unique_id, model, name, preferences)
      self.preference = preferences

  def step(self):
      super().step()

  def get_preference(self):
      return self.preference

  def generate_preferences(self, List_items):
      # see question 3
      # To be completed

 class ArgumentModel(Model):
  """ ArgumentModel which inherit from Model.
  """
  def __init__(self):
      self.schedule = RandomActivation(self)
      self.__messages_service = MessageService(self.schedule)

      # To be completed
      #
      # a = ArgumentAgent(id, "agent_name")
      # a.generate_preferences(preferences)
      # self.schedule.add(a)
      # ...

      self.running = True

  def step(self):
      self.__messages_service.dispatch_messages()
      self.schedule.step()


 if __name__ == "__main__":
  argument_model = ArgumentModel()

  # To be completed
```

2. As it was done at the end of the course 3 (Alice/Bob example), create two communicating agents named A_1 and A_2 (you may give them names if you want). These agents have an instance of `Preferences` classe. The list of items is the same for the different agents (the list of criteria may be not, you choose).

3. Write a method that allows to generate the preferences of the agent (see the example for A_1 and its corresponding ranking on criteria). You can also try to have a function that read the data from a csv file (not mandatory).

```python
def generate_preferences(self, preferences):
    """
     Set the preferences of the agent

```

```
    """
    # To be completed
```

4. Implement the following situation between the two agents:

   - A_1 to A_2 : PROPOSE(item) (no matter which one for the moment)
   - A_2 to A_1 : ACCEPT (item).

5. Update the propose/accept interaction between Agent1 and Agent2, with the following:

   - A_1 to A_2 : PROPOSE (item)
   - A_2 to A_1: ACCEPT (item) if the item belongs to its 10% most preferred item, otherwise ASK_WHY(item).

6. In case of acceptance, update the protocol to take into account the double-COMMIT interaction (see below). Both agents (A_1 and A_2) simply send a COMMITmessage to each other as soon as they have the three following information:

   - One agent made a proposal on the item;
   - The other agent accept the proposal on the item;
   - The item is available in the agent's list.

     A_1 to A_2: PROPOSE(item)

     A_2 to A_1: ACCEPT (item)

     A_1 to A_2: COMMIT (item)

     A_2 to A_1: COMMIT(item)

   After receiving the COMMIT each agent remove the item from its list of items.

   Otherwise A_1 to A_2: ARGUE()

   Warning.
   For the moment the content of this message is empty, more in the next session.

7 Generating Arguments

At the core of the dialogue between our two agents, the performative ARGUE is used to explain the reasons in favor or against a given item. **You need to check that this performative is present in our MessagePerformative.py**. As it was mentioned in the lecture, 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 $B \leftarrow A$, such that A are the premises and B a positive or negative literal, which is the conclusion.

Consider the following example:

An agent explains that it is in favor of Fuell Cell (FC) because it respects the environment. It can be encoded as follows:

$$FC \leftarrow ecology\_impact = very\_good$$

Thus the two components of the arguments are as follows:

1. Premises: we will limit ourselves to two types of propositions:

   - The value of a criterion c_i is x. This is a known fact **for both agents**, but using an ARGUE message simply asserts that this information can be used by the sender to infer a position in favor or against the item. In our communication model we shall write $c_i = x$, as in the above example.
   - The criterion c_i is preferred to the criterion c_j. 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 preference. In our communication model, we shall write $c_i \succ c_j$.

   Several premises can be used in a single message's content, all arguing in favor or against the conclusion. In that case, they are separated by commas. You can also combine the two possible types of propositions. For example:

   $$\text{ARGUE}(\neg FC \leftarrow cost = bad,\ cost \succ ecology\_impact)$$

Warning.
To generate arguments or to make a logical inference, each agent needs to have a knowledge base (KB), which contains different information, facts, domain knowledge... Due to time limitation, we restrict this knowledge base to the agent's preferences represented by the performance table (see previous session). This is clearly not sufficient to build different kinds of arguments. Moreover, each time an agent receives an argument, the agent is able to update its knowledge base to enrich its information and to update its behavior and reasoning accordingly. Again, this is not taken into account in this practical work so as to reduce the programming time.

1. In your folder you should have the package arguments as it is depicted in the following picture. This package contains three classes.

2. In the class `Comparison` implement the comparison object used in an argument object ($c_i \succ c_j$)

```python
#!/usr/bin/env python3


class Comparison:
    """Comparison class.
    This class implements a comparison object used in argument object.

    attr:
        best_criterion_name:
        worst_criterion_name:
    """

    def __init__(self, best_criterion_name, worst_criterion_name):
        """Creates a new comparison.
        """
        # To be completed
```

3. the class `CoupleValue` implement the couple value used in an argument object ($c_i = x$)

```python
#!/usr/bin/env python3


class CoupleValue:
    """CoupleValue class.
    This class implements a couple value used in argument object.

    attr:
        criterion_name:
        value:
    """

    def __init__(self, criterion_name, value):
        """Creates a new couple value.
        """
        # To be completed

```

4. In the class `Argument` implement an argument 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)

```python
#!/usr/bin/env python3

from arguments.Comparison import Comparison
from arguments.CoupleValue import CoupleValue


class Argument:
    """Argument class.
    This class implements an argument used in the negotiation.

    attr:
        decision:
        item:
        comparison_list:
        couple_values_list:
    """

    def __init__(self, boolean_decision, item):
        """Creates a new Argument.
        """
        #To be completed

    def add_premiss_comparison(self, criterion_name_1, criterion_name_2):
        """Adds a premiss comparison in the comparison list.
        """
        # To be completed

    def add_premiss_couple_values(self, criterion_name, value):
        """Add a premiss couple values in the couple values list.
        """
        # To be completed

```

5. Write a method that constructs the list of positive reasons for a proposal (an item). In our context, positive reasons will correspond to the ones for which criteria have either GOOD or VERY GOOD values for the item (see `Value` class in preferences implemented during the previous session).

```python
def List_supporting_proposal(self, item, preferences):
    """Generate a list of premisses which can be used to support an item
    :param item: Item - name of the item
    :return: list of all premisses PRO an item (sorted by order of importance based on agent's preferences)
    """
    # To be completed

```

6. Write a method that constructs the list of negative reasons against an item. They will correspond to the ones for which criteria values are either Bad or Very Bad.

```python
def List_attacking_proposal(self, item, preferences):
    """Generate a list of premisses which can be used to attack an item
    :param item: Item - name of the item
    :return: list of all premisses CON an item (sorted by order of importance based on preferences)
    """
    # To be completed

```

8 Selecting Arguments

We propose the follwoing rule to select which argument to use in our limited context of study: an agent selects the argument with the criterion with the highest position in its own preference order. You are free to propose another selection strategy if you want (be creative!) but this proposed strategy is efficient enough and simple to implement.

For instance, A_1 (see its performance table 3) has four positive arguments for the item ICED. To make a proposal, according to our rule, it will choose the argument involving Cost as it is its preferred criterion.

Consider now the following interaction:

A_1 to A_2: PROPOSE(item)

A_2 to A_1: ASK_WHY(item)

A_1 to A_2: ARGUE(item, premisses)

For the last turn the agent will use its "best" argument from its list of positive arguments (see previous question 5).

1. write a method that allows an agent to select among a list of arguments its "best" one (according to our rule or your own rule) to support a given proposal.

```python
def support_proposal(self, item):
        """
        Used when the agent receives "ASK_WHY" after having proposed an item
        :param item: str - name of the item which was proposed
        :return: string - the strongest supportive argument
        """

        # To be completed


```

2. Update and test the interaction protocol to implement the situation above.

9 Relations among arguments

1. Write a method that take as input an argument and recovers its premises and its conclusion.

```python
def argument_parsing(self, argument):
    """ returns ....
    :param argument:
    :return:
    """
    # To be completed

```

2. Write a method that decides whether an argument can be attacked or not. In our context the agent can attack or contradict an argument provided by another agent if:

   - The criterion is not important for him (regarding his order)
   - Its local value for the item is lower than the one of the other agent on the considered criteria
   - He prefers another item and he can defend it by an argument with a better value on the same criterion.
   - etc.

   Consider the following example

   A_1 to A_2: argue (ICED, c_2= good)

   A_2 may reply by

   - argue (not ICED, $c_4 \succ c_2$)
   - argue (not ICED, item c_2 = very good)
   - argue (not ICED, c_2 = bad)
   - argue(not ICED, c_4 = bas and $c_4 \succ c_2$)

   Warning.
   Of course, once it is decided that the argument can be contradicted, you need to construct its counter-argument (or to select it in your list of counter-arguments–see question 6 in section "Modeling arguments").

3. Update the interaction protocol to take into account the attack function and to implement the following situation:

   A_1 to A_2: PROPOSE(item)

   A_2 to A_1 ASK_WHY(item)

   A_1 to A_2 ARGUE(item, premisses)

   A_2 to A_1: replies with one of the previous counter-argument → ARGUE(not item, premisses) or ARGUE (new item, premisses)

4. In addition, at each ARGUE message, store in a data structure the set of exchanged arguments, such that it is possible to extract for each argument its set of counter-arguments.