1 Introduction

In this lab, you will be introduced to the field of automated negotiation. Your task will be to complete the hands-on portions of the AAAI 2022 Tutorial: Automated Negotiation: Challenges and Tools. Doing so will be an entryway for one of your final project options!

More specifically, in this lab, you will learn about a software platform used for automated negotiation called NegMAS: NEGotiation MultiAgent System / NEGotiations Managed by Agent Simulations. You will implement your own negotiation agent that runs in NegMAS.

2 Setup

You will be implementing your negotiation agents in Python. Please use your personal laptops for this lab, since the versions of Java installed in the department machines are not up-to-date.

2.1 Python and pip

First, check that you have Python and pip available from the terminal.

For Unix/macOS, you can check this by running:

\[ \text{python3 --version and python3 -m pip --version} \]

For Windows, you can check this by running:

\[ \text{py --version and py -m pip --version} \]

For more information about how to install either Python or pip, go to https://packaging.python.org/en/latest/tutorials/installing-packages/.

2.2 An IDE for Python

An Integrated Development Environment (IDE) is a software suite that consolidates basic tools required to write and test software. For example, Eclipse is an IDE for developing applications in Java. If you do not have an IDE for Python, we recommend you download Visual Studio Code: https://code.visualstudio.com/.

If you already have another IDE for Python and are comfortable using it, you may continue to use your preferred IDE.

2.3 Java

Your laptop must have Java 13 or higher for this lab. To check which version of Java you have, run on the terminal:

\[ \text{java --version} \]

If you get a message that the command java cannot be found, download Java: https://docs.oracle.com/en/java/javase/17/install/overview-jdk-installation.html.
2.4 NegMAS

To install NegMAS and the necessary negotiators for this lab, run the following command in a terminal:

- `pip install negmas`
- `negmas genius-setup`

Note: If you get an error when trying to run `negmas genius-setup` that the command `negmas` cannot be found, it is very likely that the directory that some scripts you need are not in `PATH`. Check if there was a WARNING message when you ran `pip install negmas` to find the necessary directories that need to be put in `PATH`. Then, add these directories to your `PATH`. This procedure is different for each operating system, so we suggest searching "add directory to path permanently to (your operating system here)" for guidance.

2.5 Stencil Code

Finally, download the stencil code for Lab 5 from the course website. Once you unzip the downloaded file, you should see the following Python files:

- `negotiation_runner.py`
- `lab_negotiators.py`
- `my_negotiator.py`

3 Negotiation Mechanism

The negotiation mechanism we will use in this lab is a variant of Rubinstein’s alternating offers protocol,\(^1\) called the Stacked Alternating Offers Protocol.\(^2\) It involves two agents, who take turns making offers for a finite number of rounds and/or seconds. One agent opens the negotiation with an offer, after which the other agent takes one of the following actions:

1. Accepts the offer
2. Responds with a counteroffer, thus rejecting and overriding the previous offer
3. Walks away, thus declaring an end to the negotiation, without having reached an agreement

This process repeats until either an agreement or a deadline is reached. To reach an agreement, both parties must accept the offer. If no agreement is reached by the deadline, the negotiation fails.

4 A Negotiation Scenario

Suppose there are two parties, a buyer and a seller, wanting to trade. Let’s say both parties care about:

- price, which can range from 0 to 9
- quantity, which can range from 1 to 11
- delivery time, which can range from 0 (=today) to 6

Price, quantity, and delivery time are the negotiation issues. Any combination of these three issues creates an outcome. For example, (price = 1, quantity = 4, delivery time = 6) is one possible negotiation outcome.


Now, let’s say that the preferences of each issue for the two parties are as follows:

- the buyer prefers lower prices and the seller prefers higher prices
- both the buyer and the seller prefer higher quantity
- the buyer prefers a later delivery time and the seller has a complex preference for time that goes up, then down, then up again (like a sinusoidal curve)

From these preferences, we can create preference functions for each issue. For example, the preference function of the price issue for the buyer can be represented as a linear function, \( c - p \), where \( c > 0 \) a constant, since the buyer values the outcome less as the price increases. Conversely, the preference function of the price issue for the seller can be represented as \( p \) itself, since the seller values the outcome more as the price increases. There are myriad ways of creating functions that represent these preferences.

In addition, the two parties value the issues with varying weights:

- the buyer values price and delivery time twice as much as quantity
- the seller values price and delivery time four times as much as quantity

Finally, there is a disagreement point, which is the status quo, meaning the outcome if the negotiation fails. The utility of the disagreement point is called the reservation value.

## 5 Agents’ Utilities

Let \( \Omega \) be the set of outcomes, where \((p, q, d) \in \Omega \) is an outcome with price \( p \), quantity \( q \) and delivery time \( d \). Then, if we assume that each party’s utility of an outcome is computed as the normalized weighted sum of the issue preference functions, then party \( p \)’s utility function \( u_p : \Omega \rightarrow \mathbb{R} \) can be represented as:

\[
u_p(\omega) = \frac{\sum_{i \in \{\text{price, quantity, delivery}\}} \theta_{i,p} f_{i,p}(\pi_i(\omega))}{u_p(\omega_{\text{max}})}
\]

where

- \( \theta_{i,p} \) is the weight of issue \( i \) for party \( p \)
- \( f_{i,p} \) is the preference function of issue \( i \) for party \( p \)
- \( \pi_i(\omega) \) is a selection function, which returns the value of issue \( i \) in outcome \( \omega \) (e.g., the price is 5)
- \( \omega_{\text{max}} \) is the outcome with the highest utility for party \( p \)

In particular, the buyer’s utility function \( u_b : \Omega \rightarrow \mathbb{R} \) can be represented as:

\[
u_b((p, q, d)) = \frac{2(9 - p) + 1(q) + 2(d)}{2(9) + 1(11) + 2(6)}
\]

while the seller’s utility function \( u_s : \Omega \rightarrow \mathbb{R} \) can be represented as:

\[
u_s((p, q, d)) = \frac{4(p) + 1(q) + 4(\sin \frac{2\pi d}{10})}{4(9) + 1(11) + 4(1)}
\]

**Task:** Navigate to `negotiation_runner.py` and fill in the seller’s utility function.

**Task:** Run `negotiation_runner.py`. This runs a negotiation between two pre-constructed NegMAS agents (two BoulwareTBNegotiator() agents) and plots the results of the negotiation, as in the figure below. Explain what each of the plots show. What are the blue lines? What are the red lines? What are the dotted lines? What are the horizontal lines? What are the shaded regions?
6  Experimenting with Agent Strategies

NegMAS comes with many built-in negotiating agents. For example, you used `BoulwareTBNegotiator()` as both the buyer’s and the seller’s negotiation strategy in the previous section. Some other built-in agents include `Atlas3()` and `NiceTitForTat()`.

**Task:** In the last line of `negotiation_runner.py`, change either the buyer or seller of `BoulwareTBNegotiator()` to `NiceTitForTat()` and re-run the negotiation. What changed? Play around with NegMAS a bit more, and see if you can infer `BoulwareTBNegotiator()`’s negotiation strategy?

7  Negotiation Agent Basic Structure

The base class for an agent in the Stacked Alternating Offers Protocol is already implemented in NegMAS as `SAONegotiator`. When creating new agents for the Stacked Alternating Offers Protocol, we can inherit this base class, and then implement three key functions:

- **on_preferences_changed:** This function is called when the agent’s utility function has been changed, for example, at the start of a new negotiation.
- **respond:** This function is called when an opponent has been offered an outcome, and must decide whether to accept or reject this outcome. This is where the agent’s **Acceptance Policy** is implemented.
- **proposal:** This function is called when the agent already decided to reject an offered outcome, and it now needs to create a counteroffer to send to the opponent. This is where the agent’s **Offer Policy** is implemented.

**Task:** Navigate to `lab_negotiators.py`. Look at the implementation of `BadNegotiator1()`, and run a negotiation with this agent. What is this agent’s strategy? What is bad about this strategy?

**Task:** Do the same for `BadNegotiator2()`. What is this agent’s strategy? What is bad about this strategy?

**Task:** Now, try to improve these bad agent strategies by following the **TODO**s in `ImprovedNegotiator1()` and `ImprovedNegotiator2()`.
8 Implement your Own Agent

**Task:** Implement your own agent in `my_negotiator.py`, and run two negotiations against `Atlas3()`, one where your agent is the buyer and one where your agent is the seller. Try to come up with a robust strategy that can achieve a good negotiation outcome in both cases. Your agent should have Acceptance and Offer strategies beyond what you implemented in `ImprovedNegotiator2()`.

**Task:** Change the class name of this agent from `MyNegotiator` to a unique name, and submit short descriptions of your agent’s strategies along with your `my_negotiator.py` file to this Google Form to enter the Class Competition.