Second Assignment: Message Passing

1 Introduction

In the first assignment, we worked with agents that acted independently or interacted only through their environment. In this second assignment, we focus on one of the core aspects of distributed artificial intelligence: **direct communication between agents**.

You will begin by implementing a classical message-passing multi-agent system in Python using the osBrain library. Then, you will extend your system with a Large Language Model (LLM) reasoning layer that allows agents to make decisions in a more flexible and adaptive way. Finally, you will explore agent orchestration with frameworks such as LangGraph, which coordinate multiple intelligent agents in a structured manner.

This progression—from traditional message passing to LLM-driven and orchestrated agents—illustrates the recent evolution from classical distributed AI to modern multi-agent systems.

2 Basics

We will model one of the simplest types of markets: auctions. A long-standing tradition in Catalonia is *la Subhasta del peix* (the fish auction). Different fish are displayed with an initial high price, which gradually decreases until a buyer accepts the current price. Each fish also has a bottom price; if no buyer accepts the price before reaching this value, the fish is discarded.

This type of auction is called a **Dutch auction**. We will model *la subhasta del peix* as a multi-agent system.

3 Classical Implementation

We will use Python and the osBrain library. The documentation is very clear and can be found at **Documentation**. The API reference is also available at **API Reference**.

A toy Python file, "toyAgent.py", is provided as an introduction to the library and the basic skeleton of this implementation.

3.1 Agents

You will implement two types of agents:

- **Operators**: Publishers of the products.
- **Merchants**: Potential buyers of the products.

We will have three types of fish: **[H, S, T]**, representing hake, sole, and tuna.

Each merchant has a budget, and each fish has a starting value and a bottom value. Choose reasonable numbers, and we recommend starting with equal budgets for all merchants before experimenting with different budgets.

Importantly, **each merchant has a preference for a specific type of fish**. The goal for each merchant is to obtain at least one fish of each type and as many of their preferred fish as possible. Merchants cannot have negative balances.

3.2 Parameters

Your model should allow the following parameters to be configured:

- **Number of operators**: Number of sellers participating in the auction.
- **Number of merchants**: Number of buyers participating in the auction.
- **Number of fish per operator**: Number of fish offered (can be random within a chosen range).

Note: Each merchant's preferred fish is chosen randomly.

3.3 Protocol

After setting the parameters, the message-passing protocol begins. The Operator broadcasts a new fish and its starting price to all merchants.

```
{
  operator_id: Integer,
  product_id: Integer,
  product_type: "H" | "S" | "T",
  price: Integer
}
```

If no merchant responds within a time interval:

```
{
  operator_id: Integer,
  product_id: Integer,
  msg: "BUY"
}
```

The operator decreases the price and resends the product. When a merchant buys a product, the operator sends:

```
{
  operator_id: Integer,
  product_id: Integer,
  merchant_id: Integer,
  price: Integer,
  msg: "SOLD"
}
```

If the bottom price is reached without a sale, the process restarts with a different product.

For testing, your code must log both the setup and all purchases. After simulation, output two CSV files: "setup_[date].csv" and "log_[date].csv". Files should have a format similar to:

setup_[date].csv

Merchant	Preference	Budget
1	H	100
2	S	100
3	T	100

log_[date].csv

Product	Type	Sale Price	Merchant
1	T	20	2
2	H	0	
3	H	30	1

Notice that the second product was not sold.

It is up to you to define each agent's logic for deciding when to buy. You are encouraged to be creative! If time permits, implement different strategies and compare their performance.

4 LLM-Augmented Agent Reasoning

In the first part, you implemented a classical multi-agent system based entirely on message passing: operators broadcast products, and merchants decide based on budgets, preferences, and auction price.

In this extension, you will enhance merchants by adding a **Large Language Model (LLM) reasoning layer** to guide or influence decision-making.

4.1 Goal

Each merchant now has two layers:

1. **Reactive layer** (Python + osBrain) handling:
   - Receiving product messages.
   - Maintaining local state (budget, preferences, stock).
   - Sending "BUY" messages when appropriate.
2. **Cognitive layer** (LLM) that assists the merchant in deciding whether to buy a product.

Integrate this second layer so that decision-making partially or fully depends on LLM-generated reasoning.

We recommend using the free **Polaris Alpha** model via OpenRouter. You will need to:

1. Create an OpenRouter account.
2. Obtain a free API key.
3. Call the API via Python's `requests` module.

Any model that allows **structured output** is acceptable. You may use "toyLLMAgent.py" as a starting point.

4.2 LLM Integration

When a merchant receives a product message, it formulates a prompt including:

- Current budget.
- Fish type.
- Personal preference.
- Auction price.
- Any additional relevant internal information.

The LLM must return a machine-readable answer in **JSON format**, e.g.:

```
{
  "action": "BUY" | "WAIT",
  "reason": "Short explanation of the decision"
}
```

Ensure the output is strictly parseable JSON. Most modern LLMs support this via prompt design or the `response_format` parameter (OpenRouter).

The merchant acts on the response:

- If `"action": "BUY"` and there is sufficient budget, send a BUY message.
- Otherwise, wait for the next price update.

4.3 Merchant Personalities

You are encouraged to give each merchant a distinct "personality" or strategy via the LLM prompt:

- Cautious: buys only when price is low.
- Greedy: buys quickly to beat competitors.
- Preference-driven: prioritizes favorite fish.

Encode this personality in the `system` prompt for the LLM.

5 LLM-Powered Agent Orchestration

In this part, you will implement an **agent orchestration framework** using **LangGraph**. This allows you to design multi-agent workflows where each agent has reasoning capabilities and the system coordinates interactions.

You will migrate the logic of your osBrain auction system into a graph of interacting agents:
- Nodes represent agent behaviours. - Edges define message-passing or control flow between agents.

5.1 Goal

Reimplement the auction as a LangGraph system:

- Each **Operator** and **Merchant** is a node.
- The auction protocol (broadcast, bidding, sale) is expressed as a graph.
- Each merchant retains its LLM reasoning layer.

5.2 LangGraph Overview

LangGraph is a Python library (built on LangChain) for multi-agent workflows:

- Receive inputs from other agents.
- Perform reasoning or tool use (e.g., call an LLM).
- Send structured outputs to other agents or the environment.

Documentation: **LangGraph Documentation**.

5.3 Implementation Guidelines

Create a LangGraph pipeline:

1. Define a `State` object for shared auction data: product, price, round, messages.
2. Define an **Operator** node that:
   - Broadcasts products and prices.
   - Updates price if no purchase occurs.
   - Finalizes sale when a BUY message is received.
3. Define **Merchant** nodes that:
   - Receive broadcasts.
   - Invoke LLM reasoning.
   - Return structured messages.
4. Define an evaluator node to supervise auction rounds, update prices, and continue or stop the auction.
5. Define edges representing communication: Operator → Merchants → Operator.

Refer to "toyLanggraphSystem.py" for examples.

6 Analysis

Test the system with different parameters in all three parts of the assignment. You may use a Python notebook to read CSV logs and analyze merchant performance. Compare the classical approach with the AI-augmented system.

7 Delivery and Evaluation

Deliver:

1. Two Python files (*.py): classical + LLM-augmented system, and LangGraph implementation. Executing them should produce "setup_[date].csv" and "log_[date].csv".
2. A report with comments, encountered issues, and analysis of tests.

Evaluation:

1. Quality and originality of code and models (60%).
2. Quality of analysis and experiments (40%).

Deadline: Monday, 15 December at 18:59.