Multi-Agent Systems

This project took place in the `multi-agent systems` class led by Pr. Philippe Preux at the University of Lille. It's part of the curriculum for the second year of the Machine Learning Masters' degree. Full code and explanation is here

A Multi-Agent System (MAS) is a system composed of a certain number of agents A (either a process, a robot, an ant, it depends on the problem). These agents are active in a certain environment E with and in which they interact according to certain rules (the space they can move on, the actions they can take etc.). These systems differ from deterministic systems where agents have predefined behaviours, because in a MAS agents are, at least partially, autonomous, which excludes a centralized brain. In this case, each agent has its own brain, and will listen to this brain during their "turn-of-speech" (tour de parole, french).

MAS is a common field of research in distributed artificial intelligence, in fact, MAS are a great opportunity for mass simulation of groups of people / animals, and it could almost be defined as a field of human sciences.

The aim of this project was to introduce the students with classic simulations using a particular programming paradigm : object oriented. In order to properly understant how a MAS worked we had to implement a proper architecture with an Environment, Agents and a MAS (main too but that's irrelevant). The details of these implementations are given later in this post.

We will cover 3 main simulations, each more complex than the other : Particules (particules running into each other), Wator (hunter/prey environment), Hunter (hunter "intelligence" with a pathfinder algorithm).

Backbone implementation

This project provides a basic framework for building and simulating Multi-Agent Systems (MAS). The framework is generic and highly customizable, allowing users to define their own agents and environments for a wide range of simulations. The framework consists of three main classes:

- GenericMAS: Manages the simulation and interaction between agents and the environment.
- GenericEnvironment: Represents the environment in which agents interact, including a graphical interface using Tkinter.
- GenericAgent: Defines the basic structure and behavior of an agent in the system.

Example use :

```
class MyAgent(GenericAgent):
    def decide(self, environment):
        # Define how the agent behaves each turn
        pass

class MyMAS(GenericMAS):
    def initialize_agents(self, num_agents, seed):
        # Initialize agents here
        pass
```

```
    def run_turn(self):
        # Define what happens in each turn
        pass

# Initialize the environment and the MAS
env = GenericEnvironment(width=20, height=20, cell_size=20)
mas = MyMAS(env, num_agents=10, seed=42, delay=1)

# Run the simulation for 100 turns
mas.run_simulation(100)
```

Simulating Particles in Motion with Python

1) The Principle of the Simulation

The **particle simulation** models the movement of individual agents (or particles) in a confined space, such as a 2D grid. Each particle moves autonomously, following simple rules like maintaining a set direction until hitting a boundary or another particle. Upon collisions, particles can react by changing direction, "waiting," or retrying to move in a different direction.

This simulation mimics real-world systems, such as gas molecules in a container or robotic agents navigating in shared environments. The main focus is on how particles interact with each other and their environment, emphasizing collisions, deadlocks, and movement patterns.

Key principles include:

- **Grid-based Movement:** Particles move within the confines of a grid, changing position every simulation cycle.
- **Collision Detection:** Particles detect collisions with walls or other particles and react accordingly (e.g., reversing direction or changing behavior).
- **Deadlock Management:** If two or more particles try to move into the same space simultaneously, a deadlock occurs. Various strategies can resolve these, such as random retries or directional reversals.

2) The Implementation

In Python, a particle simulation can be effectively implemented using the **Tkinter** library to create a graphical interface where particles are visualized in real time.

Key Components:

- **Particles (Agents):** Each particle has attributes like position (`posX`, `posY`), direction (`dirX`, `dirY`), and color to differentiate between agents. It moves within the grid, adjusting its position at each time step.
- **Environment:** The environment consists of a 2D grid where particles move. The environment is responsible for updating the positions of all particles and detecting collisions or deadlocks between

them.

- **GUI (Graphical User Interface):** Tkinter's `Canvas` widget is used to display the grid and visualize the movement of particles. Particles are drawn as colored shapes (e.g., circles) that move in real time, with updates occurring every few milliseconds to simulate smooth motion.

Example Code Snippet:

```
class EnvironmentGUI:
    def __init__(self, env, cell_size=50):
        self.env = env
        self.cell_size = cell_size
        self.window = tk.Tk()
        self.canvas = tk.Canvas(self.window, width=env.width * cell_size,
height=env.height * cell_size)
        self.canvas.pack()
        self.window.after(100, self.update)  # Update every 100 milliseconds
        self.window.mainloop()

    def update(self):
        self.env.update()  # Update particle positions
        self.draw_grid()    # Redraw the grid with updated particle positions

    def draw_grid(self):
        self.canvas.delete("all")  # Clear previous drawings
        for agent in self.env.agents:
            x1, y1 = agent.posX * self.cell_size, agent.posY * self.cell_size
            x2, y2 = x1 + self.cell_size, y1 + self.cell_size
            self.canvas.create_oval(x1, y1, x2, y2, fill=agent.color)  # Draw
agent as a circle

class Agent:
    def __init__(self, posX, posY, dirX, dirY, symbol, color):
        self.posX = posX
        self.posY = posY
        self.dirX = dirX
        self.dirY = dirY
        self.symbol = symbol
        self.color = color

    def update(self, env):
        # Move logic for the agent (e.g., collision detection)
        pass
```

The code above initializes an environment with a Tkinter window that continuously updates the position of agents on a grid.

3) Why is it Interesting?

Particle simulations are fascinating for several reasons:

- **Modeling Real-World Systems:** These simulations are a simplified model of physical systems like gas particles, bird flocking behavior, or even autonomous robots in swarms. Understanding how particles move and interact in a confined space can provide insights into more complex systems.
- **Agent-Based Systems:** Particle simulations are a foundation for more complex agent-based modeling, used in fields such as **robotics, computer graphics, economics**, and **biological systems**. For instance, traffic simulations, crowd behavior in urban planning, or autonomous vehicle coordination use similar principles.
- **Visual Feedback:** Unlike purely algorithmic simulations, particle simulations are **visually engaging**. By watching the agents move in real time, you can easily observe phenomena like deadlocks, collisions, or movement patterns, making it easier to spot issues or behaviors in the system.
- **Problem Solving:** Handling **deadlocks** and **collisions** is a key challenge, providing opportunities for designing intelligent behaviors or algorithms. It's not only about getting the agents to move but ensuring that they resolve conflicts and continue moving smoothly without getting stuck.
- **Interactive Learning Tool:** Visualizing particle simulations provides a **hands-on learning experience** for understanding more abstract computational concepts such as finite state machines, autonomous agents, and concurrent systems.

In conclusion, particle simulations are a captivating way to model, explore, and visualize complex systems. Whether you're simulating gas molecules, robotics swarms, or any dynamic system, these principles can help you gain a deeper understanding of how individual components behave and interact in a shared space.

Wator Simulation: Modeling Predator-Prey Dynamics

1) The Principle of the Simulation

The Wa-Tor simulation, originally conceived by A.K. Dewdney in 1984, is a simple yet fascinating model of predator-prey dynamics set in a grid-based environment resembling an aquatic ecosystem. The simulation features two types of agents:

- **Fish (Prey)**: They move around the grid, reproduce, and try to avoid predators.
- **Sharks (Predators)**: They move in search of fish to eat, reproduce after some time, but will die if they fail to find food within a certain period.

The grid represents a toroidal world, meaning it wraps around: agents that move off one edge reappear on the opposite side, creating a continuous environment. The behavior of each agent is governed by basic rules of movement, reproduction, and predation. These simple rules lead to emergent dynamics resembling real-world predator-prey cycles.

2) The Implementation

To implement the Wa-Tor simulation, we use a grid (usually a 2D array) where each cell can either be empty, contain a fish, or contain a shark. At each simulation step (or tick), the following rules are applied:

- **Fish Movement and Reproduction**:
    - Fish move randomly to neighboring empty cells.
    - After a set number of ticks, a fish reproduces by creating a new fish in an adjacent empty cell, and then continues to move.
- **Shark Movement, Feeding, and Reproduction**:
    - Sharks move randomly but prioritize moving to cells occupied by fish, representing predation.
    - After consuming a fish, the shark's energy is replenished.
    - Sharks that do not find food within a certain number of ticks will die from starvation.
    - Similar to fish, sharks reproduce after surviving for a set number of ticks, provided they have sufficient energy.

This can be implemented in Python using object-oriented principles. Each agent (fish or shark) is represented by a class with its own state (e.g., energy, reproduction counter). The environment is represented as a grid where agents interact. Each tick of the simulation moves the system one step closer to its dynamic evolution.

Here is an abstract view of the implementation process:

```
class Fish:
    def __init__(self, energy, reproduction_timer):
        # Initialization of fish attributes
        pass

    def move(self):
        # Random movement logic
        pass

    def reproduce(self):
        # Reproduction logic
        pass


class Shark:
    def __init__(self, energy, reproduction_timer):
        # Initialization of shark attributes
        pass

    def move(self):
        # Shark movement, prioritize fish
        pass

    def feed(self):
        # If a fish is nearby, shark eats and gains energy
        pass

    def starve(self):
```

```
        # Shark dies if it fails to eat in a given timeframe
        pass
```

The environment, usually represented as a grid, manages the interactions between fish and sharks:

```
class Environment:
    def __init__(self, grid_size):
        # Initialize the grid
        pass

    def update(self):
        # Move agents, check interactions, update the grid state
        pass
```

3) Why It's Interesting

The Wa-Tor simulation, while based on simple rules, generates highly dynamic and visually interesting behaviors. Its emergent patterns and cyclic dynamics closely mirror real ecological predator-prey systems, such as the relationship between wolves and rabbits or sharks and fish. By tweaking parameters such as reproduction rates, energy levels, and grid size, you can observe a wide range of outcomes—from stable cycles where predator and prey populations balance each other, to extinction events where one population dominates the other.

Here are some key insights that make the Wa-Tor simulation compelling:

- **Emergent Complexity**: Despite simple rules governing each agent, the Wa-Tor simulation exhibits complex, lifelike behaviors. Population booms, busts, and oscillations emerge naturally from the interactions between predators and prey.
- **Educational Value**: The simulation is a powerful tool for teaching concepts of ecology, population dynamics, and emergent systems. It illustrates how local interactions between individual agents can lead to global patterns, a key concept in fields ranging from biology to economics.
- **Flexibility for Experimentation**: By adjusting parameters like reproduction rates, movement speeds, and grid size, users can experiment with different ecosystem dynamics, exploring how small changes can lead to vastly different outcomes.
- **Exploration of Critical Phenomena**: The simulation can be used to explore critical points—situations where the ecosystem undergoes rapid changes, such as sudden population crashes, which can be connected to real-world phenomena like overfishing or habitat destruction.

Overall, the Wa-Tor simulation offers an engaging, interactive way to explore predator-prey dynamics and provides insight into the delicate balance that sustains ecosystems. Whether used for educational purposes or as a starting point for more complex simulations, its combination of simplicity and depth makes it an exciting subject for study.

Hunter Simulation

Principle

The Hunter Simulation is a dynamic pursuit-evasion scenario set in a grid-based environment. It features two primary agents: a Hunter and an Avatar. The Hunter's objective is to catch the Avatar, while the Avatar aims to evade capture. The simulation unfolds in a world populated with obstacles, creating a complex landscape for navigation and strategy.

Implementation

The simulation is implemented using a multi-agent system architecture:

1. **Environment**: A 2D grid where agents move and interact.
2. **Hunter Agent**: Utilizes an A* pathfinding algorithm to calculate multiple potential paths towards the Avatar. It visualizes these paths using a color gradient, with warmer colors indicating proximity to the Avatar.
3. **Avatar Agent**: Controlled by the user via arrow keys, allowing for real-time evasion tactics.
4. **Obstacle Generation**: Randomly placed walls that both agents must navigate around.
5. **Visualization**: Employs Tkinter for real-time rendering of the environment, agents, and potential paths.

Interesting Aspects

The Hunter Simulation offers several intriguing elements:

1. **Strategic Depth**: The interplay between the AI-driven Hunter and the user-controlled Avatar creates a dynamic and unpredictable scenario.
2. **Pathfinding Visualization**: The color-coded potential paths provide insight into the Hunter's decision-making process, offering a unique perspective on AI behavior.
3. **Adaptability**: The Hunter continuously recalculates its paths, demonstrating real-time adaptation to the Avatar's movements.
4. **Customizability**: Various parameters can be adjusted, allowing for diverse scenarios and difficulty levels.
5. **Educational Value**: Serves as a practical demonstration of pathfinding algorithms, multi-agent systems, and game AI concepts.

This simulation not only provides an engaging interactive experience but also serves as a valuable tool for studying and visualizing AI decision-making in complex environments.

Contributing 🤝

Contributions are highly encouraged! If you have suggestions, improvements, or feature requests, feel free to reach out to me !

License 📝

Developed by Pierre LAGUE and François MULLER at the University of Lille, France. 🚀 📊