Cops and Thieves: a Multi-Agent Game

Matteo Nestola
matteo.nestola@studio.unibo.it
Simona Scala
simona.scala6@studio.unibo.it

December 2023

The aim of this project is to simulate the Cops and Thieves game through a multi-agent system. The Cops and Thieves game involves players assuming the roles of either cops or thieves, where thieves aim to collect coins while evading capture by cops. We will employ Jason and the Q-Learning reinforcement learning algorithm to model the game's dynamics effectively.

Jason, as a platform for multi-agent system development, provides the essential infrastructure for creating intelligent agents capable of strategic decision-making and interaction. Furthermore, the Q-Learning algorithm, renowned for its ability to facilitate agents' learning from experience and optimizing their actions, aligns seamlessly with the project's goal of enabling intelligent decision-making among the game entities.

The game dynamics will involve randomly generating teams of cops and thieves, positioning safe zones and coins, and determining the playing area size based on the number of players. Thieves will navigate the playing area, including safe zones, to collect coins, while cops will patrol the area to apprehend thieves. The game concludes when either all coins are collected by the thieves, resulting in their victory, or all thieves are apprehended by the cops, leading to the cops' triumph.

1 Goal/Requirements

The project aims to develop an application centered around a multi-agent system simulating the game of cops and thieves. The Jason platform will be employed to establish the game's logic, and integration with the Q-Learning algorithm will be implemented to control the agents' movements.

The Jason platform is a powerful and flexible tool for agent-based programming. This platform offers a comprehensive set of features for designing and managing intelligent agents, facilitating the creation of complex agent interactions.

Q-Learning is a reinforcement learning technique that enables agents to learn optimal policies through trial and error. The algorithm uses a Q-table, a data structure that stores the estimated values (Q-values) of taking specific actions in specific states. These Q-values represent the expected future rewards for each action. Agents use the Q-table to make decisions. Initially, the Q-table is initialized randomly, and as the agents interact with the environment (play the game), they update the Q-values based on the rewards they receive and their experiences. Over time, with sufficient exploration and learning, the Q-values converge to optimal values, allowing agents to make the best decisions in each state to maximize their rewards.

The algorithm is well-suited for dynamic and uncertain environments, making it an excellent choice for our game simulation, where players' strategies can evolve and adapt.

1.1 Scenarios

The project in question entails the development of a simulation replicating the dynamics of a real-life gaming scenario. Users interaction is limited to initiate the simulation and subsequently to observe the unfolding game. The system systematically generates all game parameters in a randomized manner, introducing an element of unpredictability to each iteration.

2 Requirements Analysis

Specifically, we would like the game simulation to adhere to the following constraints:

- **Players**: The number of players on each team is randomly generated, ensuring a dynamic and unpredictable environment.
- **Safe Zones**: Safe places are scattered around the playing area for thieves to hide, with their number matching the total number of thieves. These safe zones are positioned randomly, adding an element of strategy as thieves must use them judiciously.
- **Playing Area**: The size of the playing area is determined by the total number of players, ensuring it is sufficiently large to accommodate safe zones and coin locations. This scalability ensures that the game remains engaging regardless of the number of participants.
- **Coins**: The number of coins corresponds to the total number of thieves and is distributed randomly throughout the playing area. This adds an element of chance and strategy to the game.
- **Starting Positions**: Both cops and thieves start at randomly generated positions within the playing area, ideally at opposite ends, enhancing the unpredictability of the game. The teams cannot see each other's starting positions, adding an element of surprise.

- **Movement Rules**: Players can move in four directions: up, down, left, and right. Thieves have the freedom to navigate the playing area, including safe zones, while cops can move freely around the playing area but are prohibited from entering safe zones. This distinction in movement rules adds depth to the gameplay.
- **Ending the Game**: The game concludes under two conditions: when all coins have been collected by the thieves (resulting in a victory for the thieves) or when all thieves have been arrested by the cops (resulting in a victory for the cops).

3 Design

3.1 Structure

The project source files were structured as follows.

```
.
├── src
│   ├── main
│   │   ├── asl
│   │   │   ├── cop_agent.asl
│   │   │   └── thief_agent.asl
│   │   ├── java
│   │   │   ├── env
│   │   │   │   ├── PlaygroundEnv.java
│   │   │   │   ├── PlaygroundModel.java
│   │   │   │   └── PlaygroundView.java
│   │   │   ├── utils
│   │   │   │   ├── Agent_g.java
│   │   │   │   ├── AgentFunctions.java
│   │   │   │   ├── Q_learning.java
│   │   │   │   ├── RandomDirectionChooser.java
│   │   │   │   └── VariableReader.java
├── copsAndThieves.mas2j
└── variables.txt
```

Specifically, the `asl` folder contains the AgentSpeak Language (ASL) files for that define the behavior of both cop agents and thief agents.

The environment in this project adheres to the Model-View-Controller (MVC) pattern, a design architectural pattern that separates an application into three interconnected components: Model, View, and Controller. In this context, the *PlaygroundModel* class serves as the Model, encapsulating the core logic and state of the environment. It defines the entities, their behaviors, and the overall rules governing the system. The *PlaygroundView* class corresponds to the View, responsible for rendering the graphical representation of the environment. It handles the visual aspects of entities, such as

drawing agents, coins, and hiding spots. Lastly, the interactions and user inputs are managed by the Controller, which in this case, is implicit within the underlying Jason framework and the interaction logic within the *PlaygroundModel*. This adherence to the MVC pattern enhances code modularity, maintainability, and extensibility. It enables a clear separation of concerns, allowing modifications to the visual representation (View) without impacting the underlying logic (Model) or the system's control flow (Controller).

Finally, a specific class has been created for Q-learning implementation, alongside several utility classes related to agent behavior and variable reading. All the variables involved (number of agents per team, number of coins, number of hiding spots, width, and height of the grid) are randomly generated inside the `start.py` file. Specifically, the number of agents per team was restricted to 4, as a higher number would have resulted in poor graphic visualization of the simulation. Starting from the randomly generated number of agents per team, the other variables are instantiated as follows:

```
n_safe_zones = n_agents_per_team
n_coins = n_agents_per_team * 2

if n_agents_per_team == 1:
   width = 5
   height = 5
else:
   if n_agents_per_team < 3:
      width = n_agents_per_team * n_agents_per_team * 2 #
          n_agents_per_team ** 3
      height = n_agents_per_team * n_agents_per_team * 2 #
          n_agents_per_team ** 3
   else:
      width = (n_agents_per_team * n_agents_per_team * 2) - 5
      height = (n_agents_per_team * n_agents_per_team * 2) - 5
```

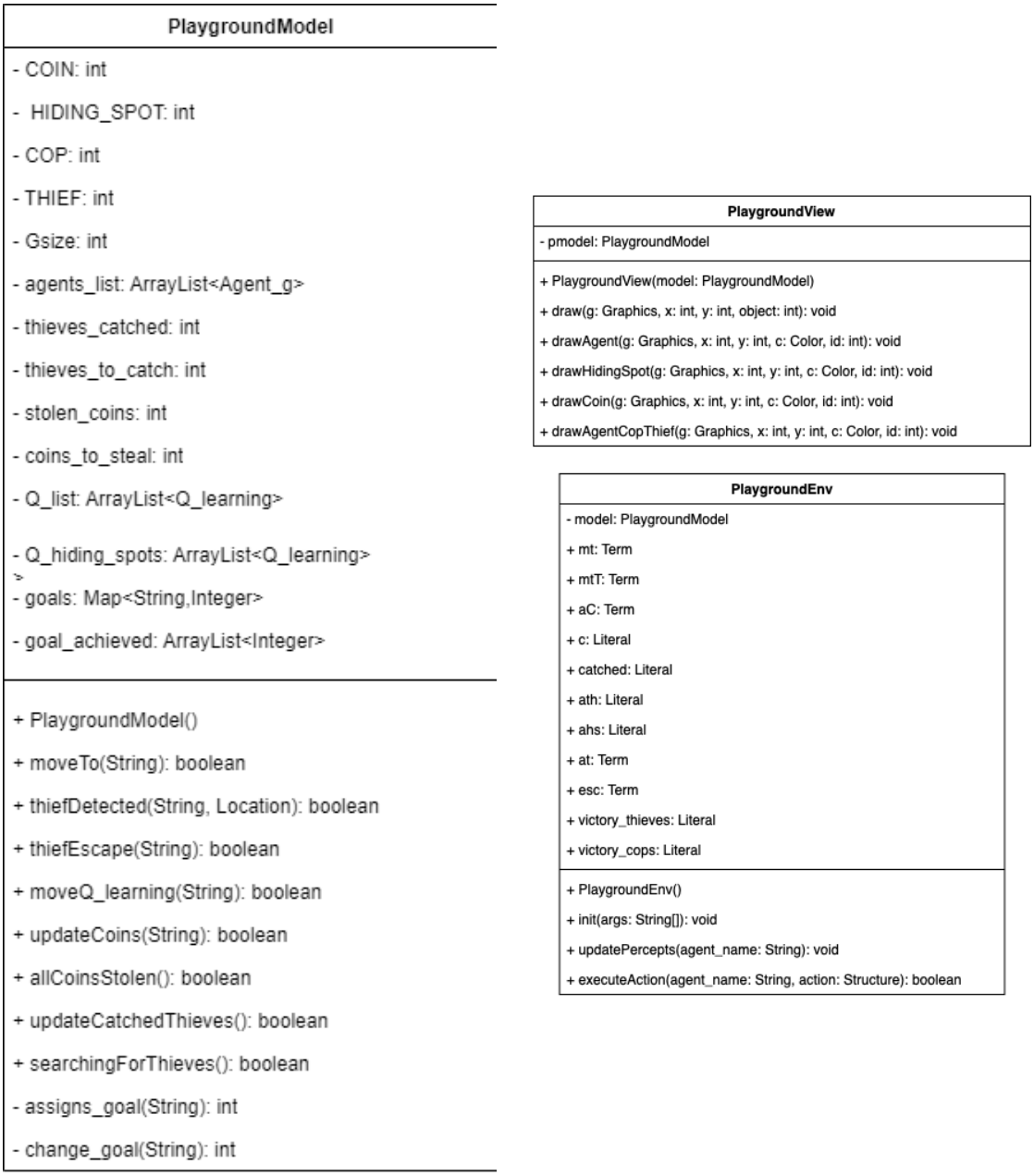


Figure 1: UML Class diagrams for MVC classes.

3.2 Behaviour

The primary goal of cops is defined by the desire to search for thieves (`!look_for(thief)`) and apprehend them. On the other hand, the thieves act with the goal of collecting all the coins (`!look_for(coin)`).

Both agents can move in four directions (up, down, left, right). Specifically, the movement of thieves is governed by Q-learning policies. This enables thieves to optimize their behavior using the Q-learning algorithm, iteratively learning and updating their movement strategies based on the rewards and penalties associated with different states in the environment. Once a goal (coin) is reached, a `change_goal` function is responsible for changing the goal (coin) that a thief is assigned to reach. Before finalizing the decision, it checks whether the chosen goal is already assigned to another agent or has been achieved by any agent. If the goal isn't suitable, it goes back to the drawing board, making sure the thief gets a goal that makes sense for the current state of the environment. The function checks if all but one coin goals have been achieved as well. If that's the case, it returns `-1`, signaling that there are no more viable goals left for the agent.

The Q-learning approach could not be implemented for the cops because the algorithm is trained on fixed positions of the objectives to reach, and cops must apprehend thieves that are constantly in motion.

However, the cops were equipped with a behavior that notifies all other cops through a broadcast message `.broadcast(tell, detected_thief(_,_,_));` when a thief is in the neighborhood of a cop, i.e., within a certain Manhattan distance (`manhattanDistance <= 2`). Upon `thiefDetected`, all the cops will get to the location where the thief was spotted. On the other hand, a thief will escape (`thiefEscape`) by determining a new location always based on Q-learning policies.

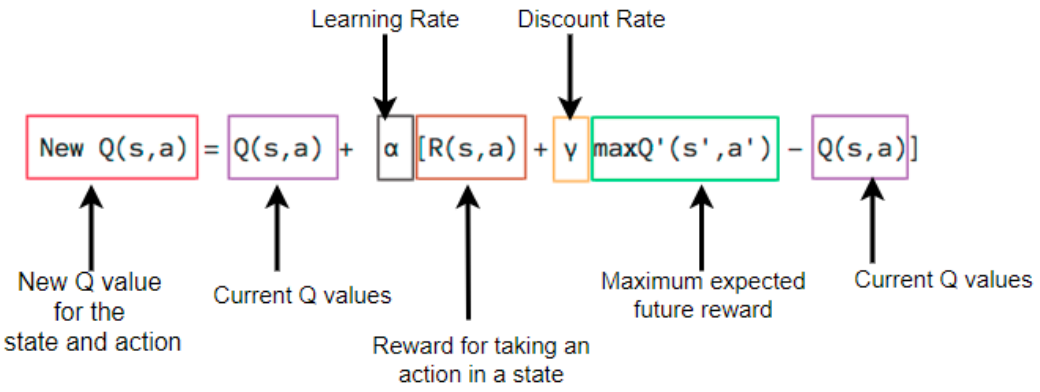
Through the grid, thieves stop in the safe zones when they land on it (`at(hiding_spot)`). In this case, an internal timer counts a total of 10 seconds after which the thief exits the safe zone.

3.3 Interaction

The `Q_learning.java` class interacts with the environment to guide the behavior of thieves. Specifically, thieves start with an initial Q-matrix (`Q`) that represents the expected cumulative rewards for executing specific actions in different states. The Q-matrix is initialized with values from the corresponding positions in the reward matrix (`R`).

Thieves undergo a training process in which they explore the environment, engage with the surrounding states, and update their Q-values based on the outcomes of their actions. The training process includes selecting a random initial state and iteratively taking actions based on a balance of exploration and exploitation. Thieves randomly choose actions with a certain probability, enabling the exploration of new paths in the environment. Simultaneously, they exploit learned information by favoring actions with higher Q-values, reflecting the expected cumulative rewards.

The Q-values are updated using the Q-learning formula:



Here, `alpha` is the learning rate, determining the weight given to new information, and `gamma` is the discount factor, balancing immediate and future rewards. `R(state, action)` represents the immediate reward associated with taking the specified action in the current state, while `Max(next state, all actions)` denotes the maximum Q-value among the possible next states and actions. Thieves receive positive rewards for moving toward coins and negative penalties for encountering obstacles or moving away from coins.

By experiencing and learning from these rewards and penalties, thieves adapt their behavior to maximize cumulative rewards over time.

4 Self-assessment / Validation

The software satisfies all the requirements above.

However, as previously mentioned, it was not possible to implement Q-learning-based movement for the cops.

Additionally, when executed, it appears that the simulation does not stop correctly when the cops win, whereas it does when the thieves win.

Lastly, the graphical environment does not seem to be fluid, making it challenging to visualize the simulation easily.

5 Deployment Instructions

Before cloning the Git repository of the project, the following dependencies need to be installed:

* `Java v.19.0.2`
* `Gradle v.1.2.1`
* `Jason v.3.1.2`
* `Python v.3.9.6`

Once the dependencies have been correctly installed, one can proceed to clone the Git repository following the instructions below.

Open a terminal or command prompt on the local machine. Navigate to the desired directory for repository cloning, utilizing the `cd` command to switch directories. For instance:

```
cd path/to/your/directory
```

The following command should be executed to clone the repository:

```
git clone https://gitlab.com/pika-lab/courses/mas/projects/
mas-project-nestola-scala-ay2223.git
```

The command will download the repository to the local machine. If the repository is private, users might be prompted to enter their GitLab username and password or use an access token.

```
Username for 'https://gitlab.com': your_username
Password for 'https://your_username@gitlab.com':
```

Credentials should be entered to proceed.
Once the cloning process is complete, proceed to enter the cloned directory.

```
cd mas-project-nestola-scala-ay2223
```

Execute the following command to initiate the build process:

```
python3 start.py
```

This command will automatically trigger the Gradle build and set up the necessary variables for the project. The simulation can be observed in the graphical window titled *Cops and Thieves*.

6 Usage Examples

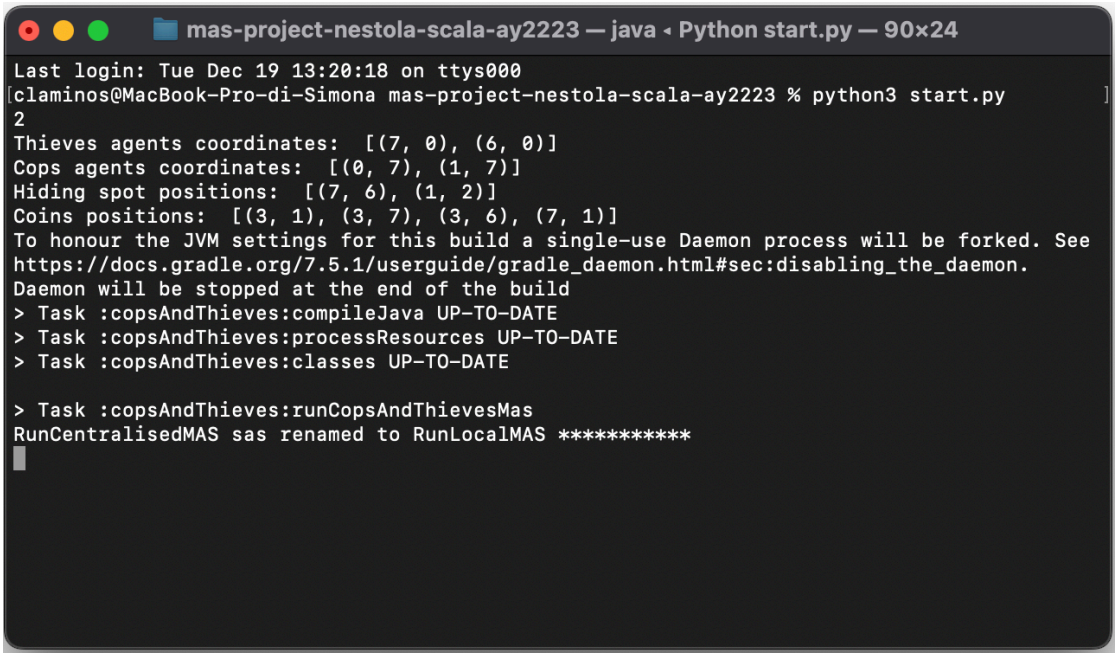
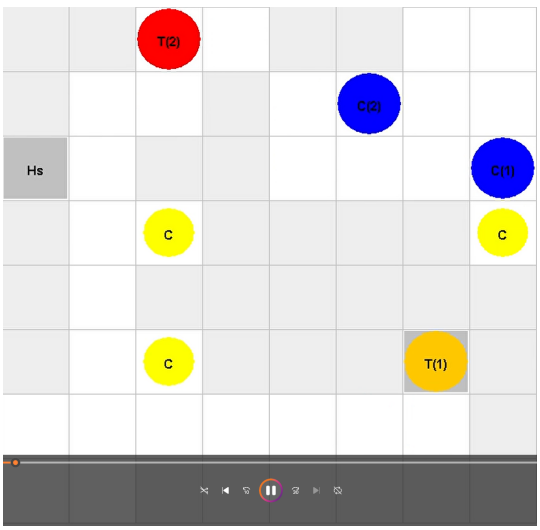
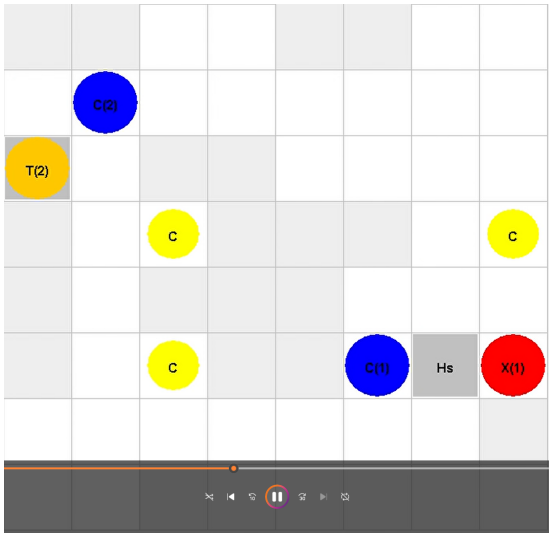


Figure 2: Execution of the start command in the shell.





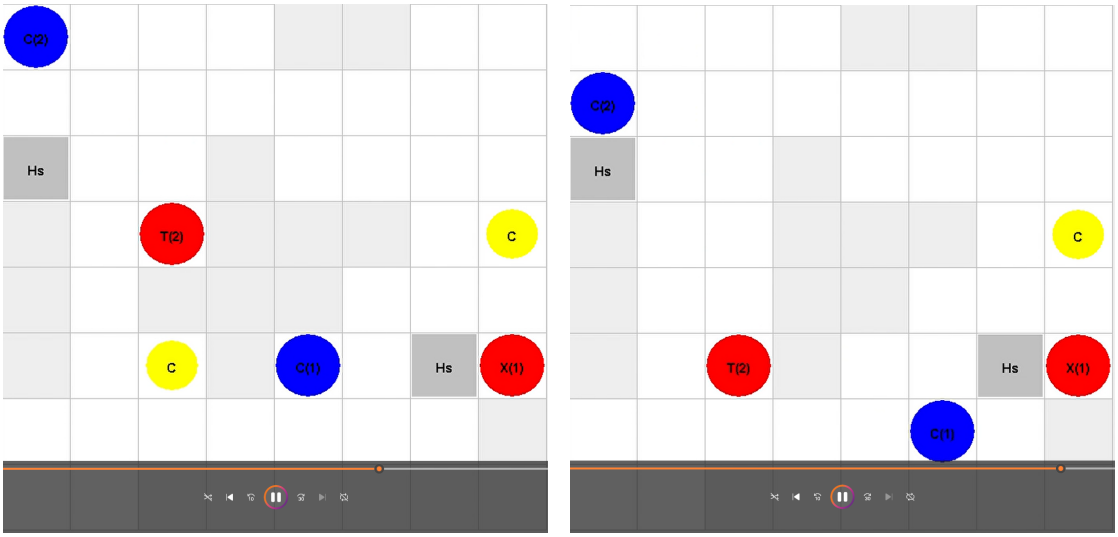


Figure 3: Screenshots from the simulation.

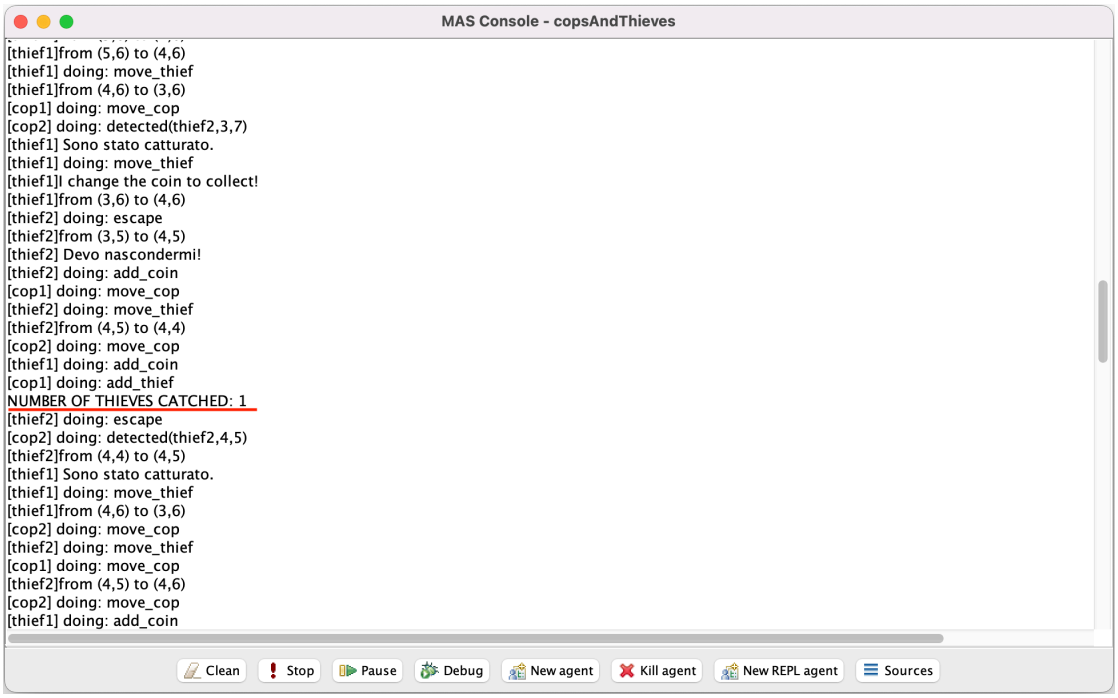


Figure 4: Count of captured thieves.

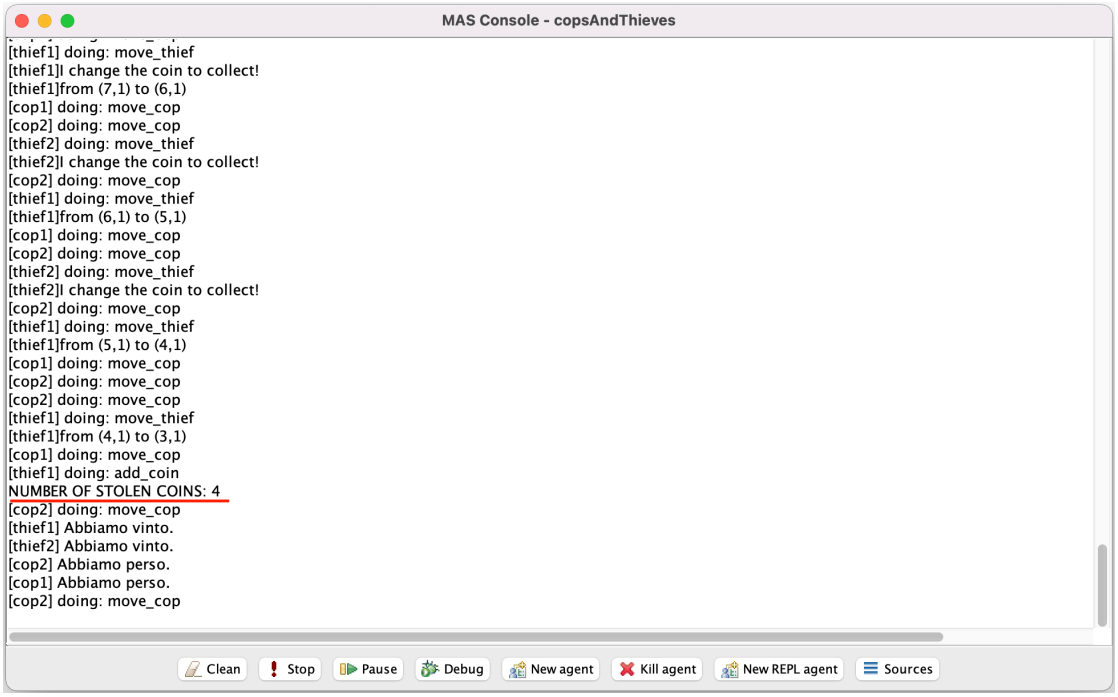


Figure 5: Count of stolen coins.