Jason-Based Multi-Agent System for City Safety: A Simulation of Agents Cooperating to Arrest Criminals

Giulia Benvenuto

s4678610@studenti.unige.it

1. Introduction

The objective of this project is to use Jason which is an interpreter for an extended version of AgentSpeak, that provides a platform for the development of multi-agent systems in order to create a dynamic simulation of an environment representing a city where police agents, civilian agents and clue agents with their different behaviours, cooperate to discover and arrest criminals.

2. City Safety Problem

To understand why a Multi-Agent System (MAS) is well suited for addressing the city safety problem, it's essential to analyse the problem at its core.
Safety is a key issue in city areas, affecting the happiness and the satisfaction of the city's residents, for these reasons is reasonable to think that civilians living the city will collaborate with law enforcement, not only to enhance their personal safety but also to contribute to the overall security of the environment.
This collaboration typically involves sharing information useful to quickly find a criminal and arrest him as soon as possible.

3. Harnessing Multi-Agent Systems to solve the City Safety Problem

After the analysis of the general problem, we can easily understand why a simulation about city safety can be well represented by a MAS.
The inherent collaboration and cooperation exemplified in the city safety problem (between civilians, police, and eventually other agents) perfectly aligns with the capabilities of MAS, that generally are designed to simulate and manage complex interactions among multiple autonomous agents; moreover, they are particularly adept in simulations where the agents are working to achieve a shared and common goal.
Finally, leveraging the distributed intelligence and collaborative nature a MAS provides a good framework to demonstrate the cooperative effort of agents working together within the city to achieve safety.

4. Development Framework

To implement the multi-agent simulation for city safety I used a development framework that integrates multiple technologies, ensuring that the interaction between agents can also be visually tracked through a Graphical User Interface (GUI).
This GUI displays the dynamic movements of agents within the city environment, represented by a grid of cells, allowing for a more intuitive understanding of the agents' position, of their communications and collaborative behaviours.

4.1 Jason

As outlined on the Jason official website: "Jason is an interpreter for an extended version of AgentSpeak. It implements the operational semantics of that language and provides a platform for the development of multi-agent systems." [1]
Thereby, in this project, Jason's capabilities define and manage the behaviours of four distinct types of agents, each dynamically interacting within the city environment. These agents – police agents, civilian agents, clue agents and criminal agents – utilize Jason to manage their beliefs, intentions, goals and act accordingly.
This implementation allows each agent type to operate based on a distinct set of perceptions and objectives that influence their movement and interactions in the simulation.

4.2 Java

A portion of the project is developed in Java, stemming from the fact that Jason is built on Java. This aspect allows Jason to integrate with and extend Java's capabilities.
This integration is crucial for Jason as it allows the enhancement and expansion of the functionalities of the Multi-Agent system, such as adding percepts dynamically to agents as they navigate the city.
In my project this integration is evident in the structure of three critical components: "CityEnvironment.java", "CityModel.java" and "CityView.java" that extend Java-based classes specifically designed for Jason's environment. More precisely they respectively extend "Jason.environment.Environment" [2], "GridWorldModel" [3] and "GridWorldView" [4].

5. Environment and Agents Design and Implementation

In the following section I will provide a detailed description of the implementation of both the city environment and the agents in the project.
This section will also include an in-depth examination of the most important files in the program, highlighting how each component contributes to the overall functionality and outcome of the system.

5.1 Run the code – crminals_and_agents.mas2j

To run the program move into the "MultiAgent_Systems" folder and run the following command: `./jason criminals_and_agents/criminals_and_agents.mas2j`

```
13  MAS criminals_and_agents {
14
15      environment: city.CityEnvironment(1)
16
17      agents:
18          police #3;
19          criminal #2;
20          clue #4;
21          civilian #4;
22  }
```

This is the core of the program since in it there is the configuration of the components of the MAS: environment and agents.

- `environment: city.CityEnvironment(1)`
  is the setup of the environment calling the Java class responsible of it.
- `agents: police #3; criminal #2; clue #4; civilian #4;`
  is the setup of the agents that will populate the environment. Where the hashtag and the following number define how many instances of each agent should be initialized.

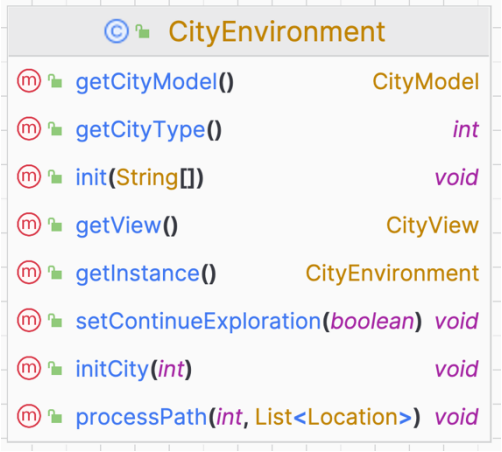
5.2 Environment

The City Environment in this Multi-Agent system serves as the dynamic stage where agents perform their roles and interact. It is implemented through a structured architecture that involves the use of three Java classes that extend Jason classes: CityEnvironment, CityModel and CityView.
Each one of these components have a distinct role in defining, modelling, and visualizing the environment and the agents in it.
Together, they aim to create an environment filled with agents and obstacles, mimicking real-world cities.

5.2.1 CityEnvironment.class

This is the Java class utilized by the .mas2j file to instantiate the city environment where the agents interact and navigate. It extends ”Jason.environment.Environment” that serves as the backbone for the dynamic space of the simulation offering some methods to manage agents such as: `addPercept()` or `removePercept()` (entire list of methods in the API documentation of the class [2]).
Such methods can also be overridden if needed to define specific behaviours.

To understand the responsibilities of this class we can look at the UML diagram:



The class contains method to get or set variables and instances and the most critical and important methods are:

- `init(String[])`: that is fundamental for starting up the environment.
- `processPath(int agId, List<Location> path)`: that handles the movement of police agents within the city grid following a path computed by the A* algorithm.

5.2.2 CityModel.java

This is the Java class that manages the creation of the grid representing the city, the state and the evolution of it during the simulation. This class extends “GridWorldModel” [3] from the Jason environment framework enabling the usage of the provided grid functionalities.
With this class we can associate a 40x40 grid to the environment and put in it all the dynamic and static entities:

1) **Obstacles**: static objects, such as walls and houses that define the physical constraints of the simulation.

 Wall obstacle  House obstacle

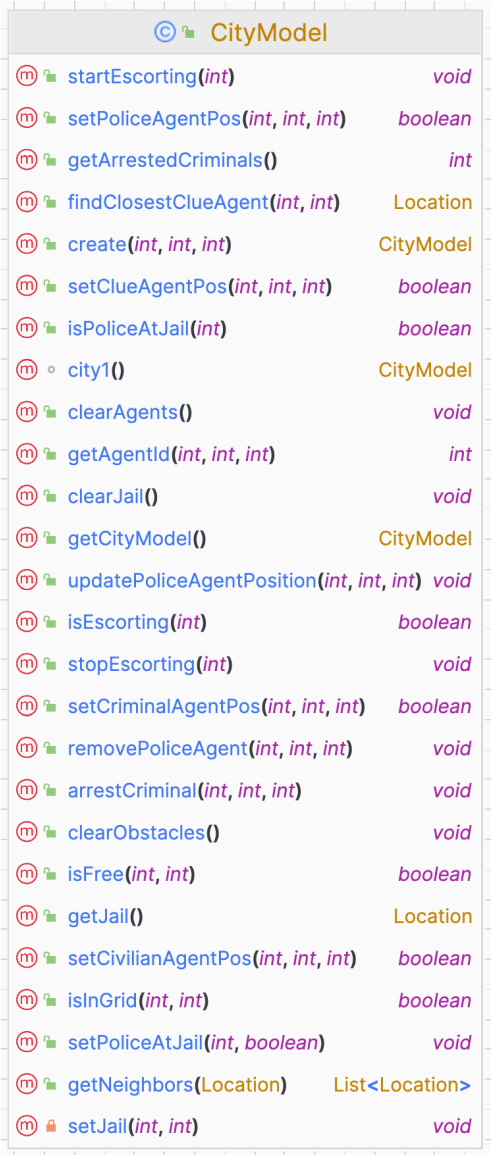
2) **Jail**: static and designed location in the grid where arrested criminals are held. It changes icon depending on the number of arrested criminals.

 Free jail  One criminal in jail  Two criminals in jail

3) **Agents**: of different types, having different roles and capabilities. Police agents are moving in the grid exploring it, other agents are fixed in some positions.

 Police  Civilian  Clue  Criminal

As before, to understand the responsibilities of this class we can look at the UML diagram:



This class includes various methods. Before each method in the code, there is a comment explaining what the method does.

However, following I will describe the most important methods of the class:

- `city1()`: creates the city model, set in it all the agents by calling the methods "setNameofagentPos()", set the position of the jail, set the position of obstacles.
- `setPoliceAgentPos()`: set the position of the police agents and add them the percepts about their own position and id, the start position, the end position and jail position.
- `setCivlianAgentPos()`: set the position of the civilian agents and add them the percepts about their own position, id and the information about the closest clue agent.
- `setClueAgentPos()`: set the position of the clue agents and add them the percepts about their own position, id and the information about criminal position X or Y.

- `setCriminalAgentPos():` set the position of the criminal agents and to add them the percepts about their own position and id.

Then there are some methods to handle and track the state and the position of police agents like:
`isEscorting(int), startEscorting(int), stopEscorting(int), isPoliceAtJail(int), updatePoliceAgentPosition(int, int, int), removePoliceAgent(int, int, int).`

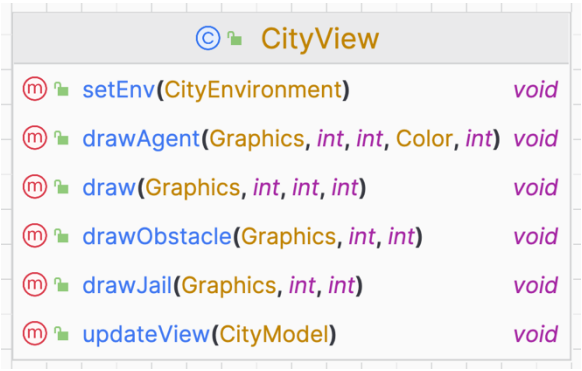
There are some methods are used for environmental manipulation such as:
`setJail(int, int), getJail(), clearJail(), clearObstacles()` and `clearAgents().`

And there are some utility methods:
`getNeighbors(Location), findClosestClueAgent(int, int), isInGrid(int, int), ifFree(int, int)`

5.2.3 CityView.java

This is the Java class designed to visualize the city environment and draw on it the agents and the obstacles. This class extends the "GridWorldView" [4] class of the Jason environment framework that provides all the view components for a GridWorldModel and methods to retrieve the width and the weight of the grid used.
So, this class is structured to manage and update the visual representation based on changes in the model.

As before, to understand the responsibilities of this class we can look at the UML diagram:



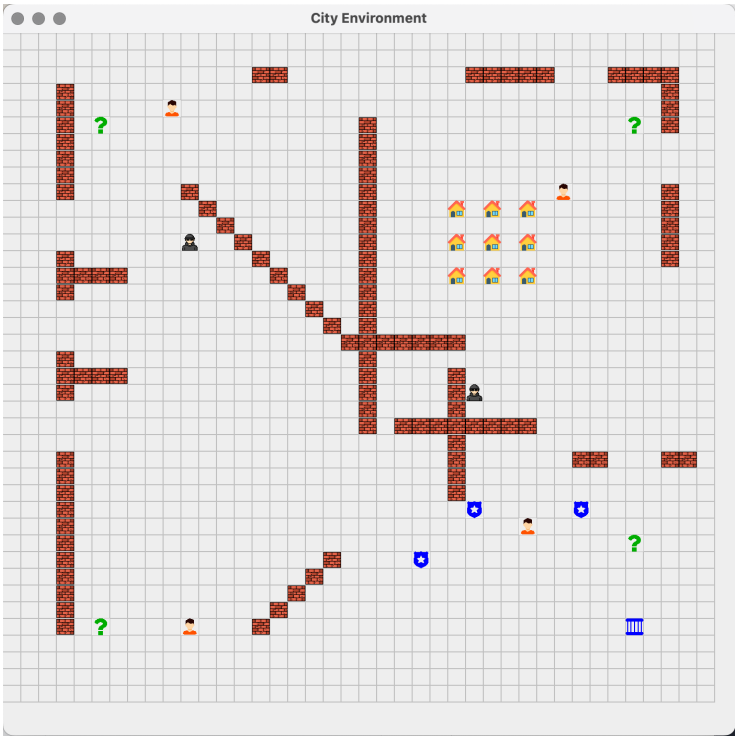
The class contains methods to draw:

- `draw(Graphics, int, int, int)`: is the overridden method of the GridWorldView Jason class used to draw objects (obstacles and jail).
- `drawAgent(Graphics, int, int, Color, int)`: is the overridden method of the GridWorldView Jason class used to draw the agents.

- `drawObstacles(Graphics, int, int)`: is the method used to draw the obstacles with their icon (walls and houses).
- `drawJail(Graphics, int, int)` : is the method used to draw the jail with its icon.
- `updateView(CityModel)`: is the method used to update the visualization of the city environment every time is needed.

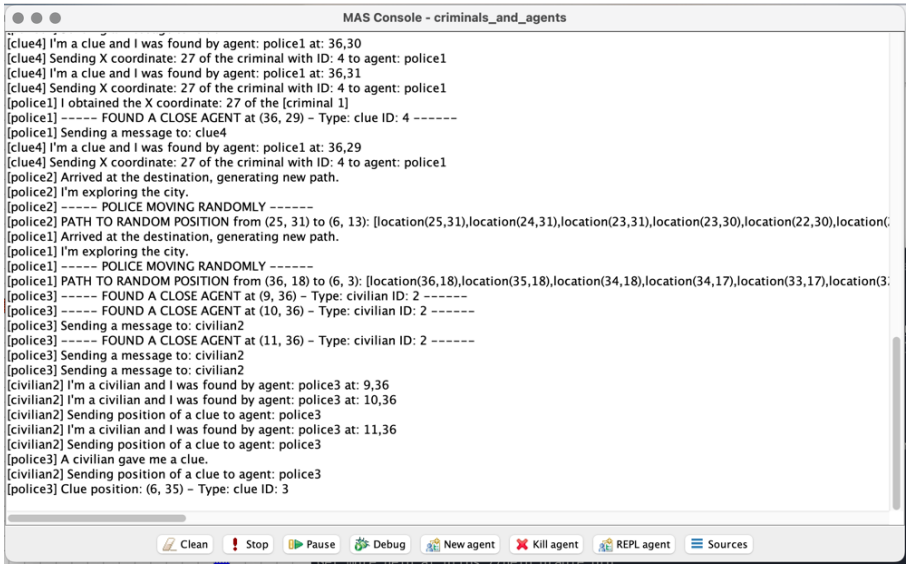
5.2.4 City Environment Result

Using the components described in the previous sections, the city environment is visualized in a window that illustrates the dynamic movements of the police agents and statically represents the other agents and the obstacles.



In addition to the graphical representation of the city there is also the MAS console. This console offers a more detailed report of the agents' behaviours and interactions:

- **Agent path tracking:** the console displays the paths of the police agents as a series of (x, y) coordinates. This gives a more comprehensive understanding of the police agent navigation within the grid from a start to an end position.
- **Agent communications:** the console captures and displays the interactions between agents, including the exchange of messages and information. This is crucial to understand the coordination and the cooperativity of agents in order to achieve their objective.



5.3 Agents

The city environment is populated with four types of agents: police, civilian, clue and criminal. Each agent has different beliefs and plays a different role in the dynamics of the simulation.
The lifecycle of all of them starts when they are created and ends when the police agents arrest both the criminals and took them to prison.

```
[police3] I'm a police agent.
[criminal1] I'm a criminal.
[clue4] I'm a clue.
[criminal2] I'm a criminal.
[clue1] I'm a clue.
[clue3] I'm a clue.
```

```
[police1] Agent with name: police1 has been destroyed.
[police1] Agent with name: police1 has been destroyed.
[clue2] Agent with name: clue2 has been destroyed.
[clue1] Agent with name: clue1 has been destroyed.
[civilian2] Agent with name: civilian2 has been destroyed.
[clue4] Agent with name: clue4 has been destroyed.
[police2] Agent with name: police2 has been destroyed.
[civilian3] Agent with name: civilian3 has been destroyed.
[civilian1] Agent with name: civilian1 has been destroyed.
[civilian4] Agent with name: civilian4 has been destroyed.
[clue3] Agent with name: clue3 has been destroyed.
[criminal1] Agent with name: criminal1 has been destroyed.
[criminal2] Agent with name: criminal2 has been destroyed.
```

5.3.1 Police Agent

- **Overview:** in the city environment, three police agents are deployed, representing the most complex and dynamic elements within the system. These agents are tasked with navigating and exploring the grid-based city. Their primary objective is to detect and interact with other agents, particularly focusing on identifying and apprehending criminals.
  Their initial positions are:

```
// Police agents
city_model.setPoliceAgentPos(0, 35, 34);
city_model.setPoliceAgentPos(1, 34, 35);
city_model.setPoliceAgentPos(2, 34, 34);
```

- **Behaviour:**
    - Start exploration from jail.

```
[police2] ----- POLICE MOVING RANDOMLY ------
[police2] PATH TO RANDOM POSITION from (34, 35) to (34, 23): [location(34,35),location(33,35),location(33,34),location(33,33),location(34,33),location(34,32),
[police3] PATH TO RANDOM POSITION from (34, 34) to (36, 16): [location(34,34),location(34,33),location(35,33),location(35,32),location(35,31),location(36,31),
[police1] PATH TO RANDOM POSITION from (35, 34) to (14, 38): [location(35,34),location(35,33),location(34,33),location(33,33),location(33,34),location(33,35),
```

    - Compute the shortest path to a random unoccupied position within the grid using the A* algorithm [5].
    - During the movement, at each step the agent looks around itself, checking the 8 neighbouring cells.
    - If all the neighbouring cells are free the agent proceeds along the path, if in the neighbouring cells there is another agent communicate with it.
    - If the agent found is a civilian, the police agent gets the (x, y) coordinates of a clue agent. At the next iteration the end position will be the one of the obtained clue.

```
[police3] ----- FOUND A CLOSE AGENT at (28, 30) - Type: civilian ID: 4 ------
[police3] Sending a message to: civilian4
[civilian4] I'm a civilian and I was found by agent: police3 at: 28,30
[civilian4] Sending position of a clue to agent: police3
```

    - If the agent found is a clue, the police agent gets either the x or the y of a criminal. If both x and y of the same criminal has been found, at the next iteration the end position will be the one of the criminals.

      Clue has been found:

```
[police2] ----- FOUND A CLOSE AGENT at (34, 31) - Type: clue ID: 4 ------
[police2] Sending a message to: clue4
[clue4] I'm a clue and I was found by agent: police2 at: 34,31
[clue4] Sending X coordinate: 27 of the criminal with ID: 4 to agent: police2
[police2] I obtained the X coordinate: 27 of the [criminal 1]
```

      Both x and y of the same criminal has been found:

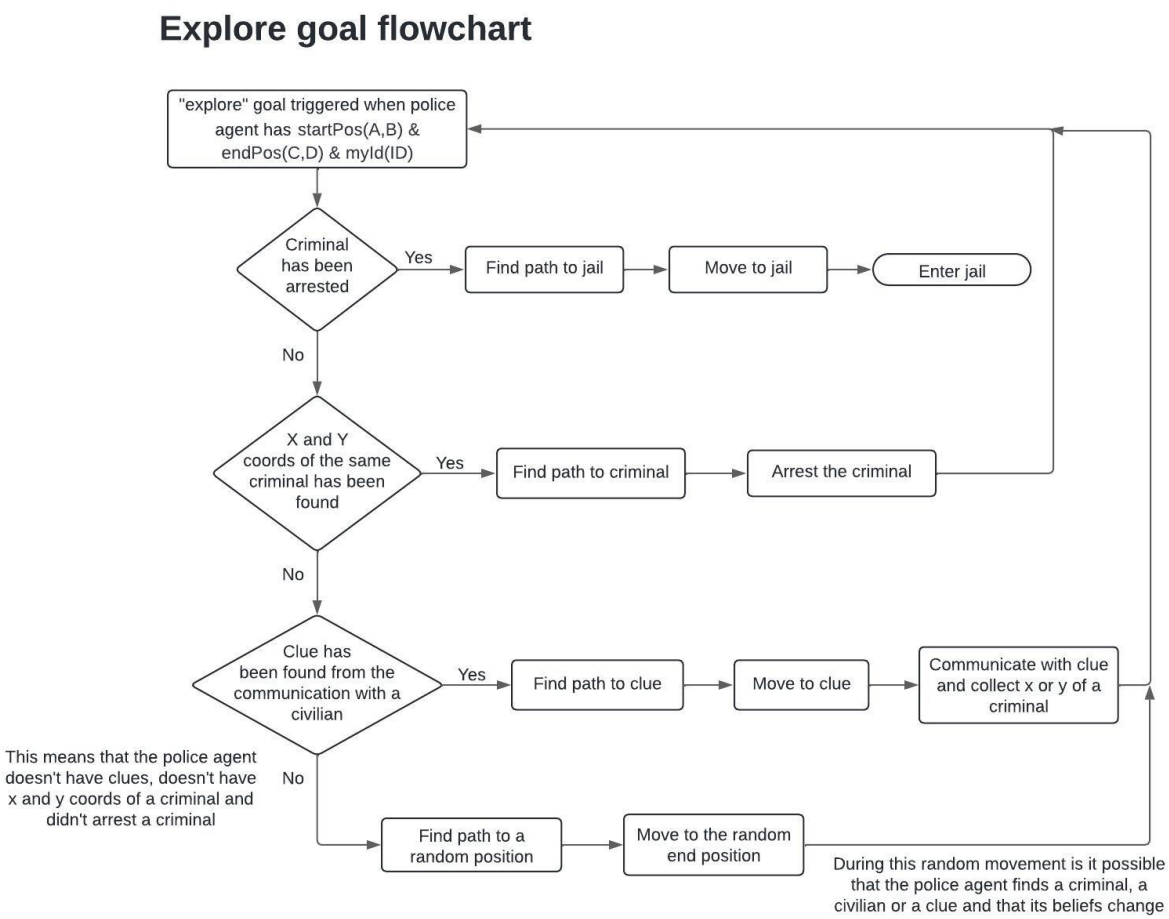
```
[police3] ----- FOUND A CLOSE AGENT at (6, 5) - Type: clue ID: 1 ------
[police3] Sending a message to: clue1
[police3] Sending a message to: clue1
[clue1] I'm a clue and I was found by agent: police3 at: 6,6
[clue1] Sending Y coordinate: 12 of the criminal with ID: 3 to agent: police3
[clue1] I'm a clue and I was found by agent: police3 at: 6,4
[clue1] I'm a clue and I was found by agent: police3 at: 6,5
[clue1] Sending Y coordinate: 12 of the criminal with ID: 3 to agent: police3
[clue1] Sending Y coordinate: 12 of the criminal with ID: 3 to agent: police3
[police3] I obtained the Y coordinate: 12 of the [criminal 1]
[police3] I'm exploring the city.
[police3] ----- POLICE HAS BOTH X and Y of CRIMINAL ------
[police3] ----- GOING TOWORDS CRIMINAL ------
[police3] PATH TO CRIMINAL from (5, 2) to (11, 12): [location(5,2),location(6,2),location(7,2),
```

- If the agent found is a criminal, broadcast all the other agents communicating that a criminal has been arrested and escort him to the jail, when reached enter it.

```
[police1] ----- YOU ARE A CRIMINAL AND YOU ARE UNDER ARREST AT: (26, 21) ------
[civilian2] I feel safer! Tank you agent: police1
[civilian1] I feel safer! Tank you agent: police1
[civilian4] I feel safer! Tank you agent: police1
[police2] Well done collegue: police1 you did a great job.
[civilian3] I feel safer! Tank you agent: police1
[criminal2] Police agent is picking me up at: 27,20 and taking me to jail.
```

- If the agent receives a broadcast message from another police agent congratulate him.
- When the end position of the path is reached start again.

The behaviour outlined is fully encapsulated within the "police.asl" script, where the implementation of the "explore" goal is crucial to define the logic that governs the actions of the police agent, for this reason I represented it through the following flowchart.



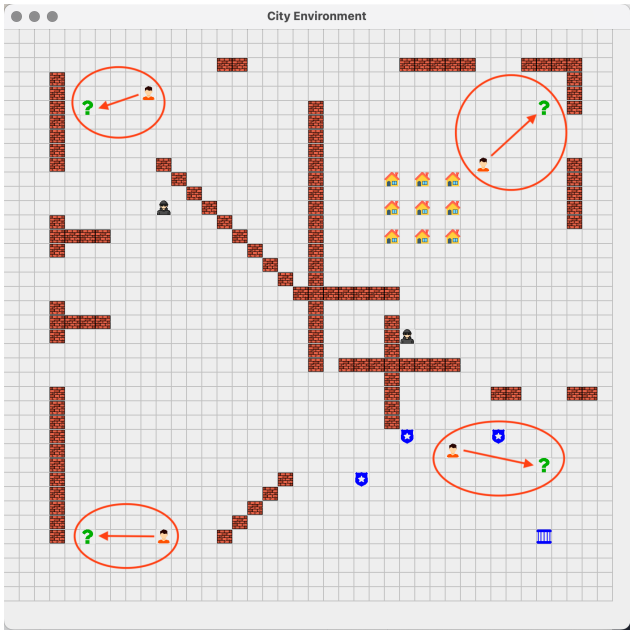
5.3.2 Civilian Agent

- **Overview:** in the city environment, four civilian agents are deployed.
  These agents are still in predefined positions and during the simulation they do not move.

```
// Civilian agents
city_model.setCivilianAgentPos(9, 9, 4);
city_model.setCivilianAgentPos(10, 10, 35);
city_model.setCivilianAgentPos(11, 31, 9);
city_model.setCivilianAgentPos(12, 29, 29);
```

  Their primary objective is to inform police agents about the position of their closest clue saved as a belief in the civilian agent.

  In the following image the relation between civilian and clue is highlighted.



- **Behaviour**:
  - Stay still in a position.
  - When it is found by a police agent, communicate with him, sending the coordinates of the closest clue.

```
[civilian2] I'm a civilian and I was found by agent: police2 at: 9,35
[civilian2] I'm a civilian and I was found by agent: police2 at: 9,36
[civilian2] Sending position of a clue to agent: police2
[civilian2] Sending position of a clue to agent: police2
```

  - Upon receiving a broadcast message from a police agent about the arrest of a criminal, respond with a message thanking him.

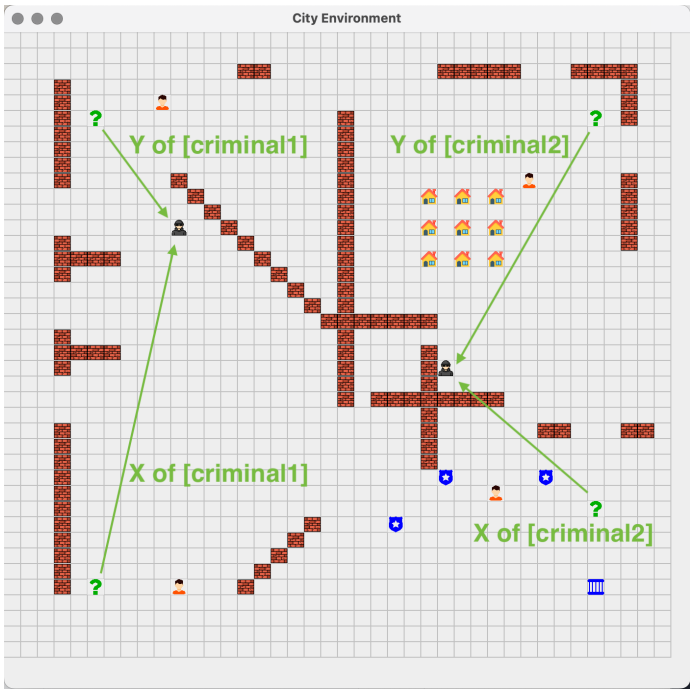
```
[civilian2] I feel safer! Tank you agent: police1
[civilian1] I feel safer! Tank you agent: police1
[civilian4] I feel safer! Tank you agent: police1
```

5.3.3 Clue Agent

- **Overview:** in the city environment, four clue agents are deployed.
  These agents, like the civilians, are still in a predefined position and during the simulation they do not move.

```
// Clue agents
city_model.setClueAgentPos(5, 5, 5);
city_model.setClueAgentPos(6, 5, 35);
city_model.setClueAgentPos(7, 35, 5);
city_model.setClueAgentPos(8, 35, 30);
```

  Their primary objective is to inform police agents about the X or Y coordinate of a criminal. The two clues in the left part of the city are relative to the [criminal1] that is positioned in the left part of the city, the two clues in the right part of the city are relative to the [criminal2] that is positioned in the right part of the city.



- **Behaviour**:
  - Stay still in a position.
  - When it is found by a police agent, communicate with him, sending the known X or Y coordinate of the criminal with the ID of the correspondent criminal.

    [clue4] I'm a clue and I was found by agent: police1 at: 36,29
    [clue4] Sending X coordinate: 27 of the criminal with ID: 4 to agent: police1
    [police1] I obtained the X coordinate: 27 of the [criminal 1]

5.3.3 Criminal Agent

- **Overview:** in the city environment, two criminal agents are deployed.
  They are the simplest type of agent since they are still in the city grid mimicking the fact that they are hiding.

```
// Criminal agents
city_model.setCriminalAgentPos(3, 10, 12);
city_model.setCriminalAgentPos(4, 26, 21);
```

- **Behaviour**:
  - Stay still in a position.
  - When it is found by a police agent it receives a message from it saying that it is under arrest.

```
[criminal1] I'm a criminal and I was found by agent: police1 at: 11,12.
[criminal1] Police agent is picking me up at: 11,12 and taking me to jail.
[police1] ----- YOU ARE A CRIMINAL AND YOU ARE UNDER ARREST AT: (10, 12) ------
```

  - When apprehended by a police agent, the criminal agent is removed from the grid as they are escorted to jail by the police agent who arrests it.

6. Internal Actions

Internal Actions serve as a building block for agents to perform operations that go beyond basic logical reasoning. To develop an Internal Action, one must extend a java class with the "DefaultInternalAction" class provided by Jason [6]
This approach allows the implementation of specific agent behaviours, in my project the most important Internal Actions are the following.

6.1 FindPath.java Internal Action

This Internal Action handles the logic of finding a path between two locations (x, y) in the city environment and it is used by the police agents in the "explore" goal in order to find the shortest path from a starting position to an ending position using the A* algorithm. Finally is also used to move the police agents' icons in the grid.

- The Internal Action in the Jason police.asl file is called like this:

```
path.FindPath(ID, A, B, C, D, Path);
```

  ID: police id
  A, B: (x, y) coordinates of the start position
  C, D: (X, y) coordinates of the end position
  Path: variable to save the computed path from (A, B) to (C, D)

- The first thing done in the java file of the Internal Action is to save these values:

```
int policeId = (int)((NumberTerm) args[0]).solve();     // Get police agent ID
int startX = (int)((NumberTerm) args[1]).solve();       // Get start X coordinate
int startY = (int)((NumberTerm) args[2]).solve();       // Get start Y coordinate
int endX = (int)((NumberTerm) args[3]).solve();         // Get end X coordinate
int endY = (int)((NumberTerm) args[4]).solve();         // Get end Y coordinate
```

- Then the A* algorithm is used to compute the path:

```
// Create an instance of AStar to perform pathfinding
AStar aStar = new AStar(cityModel);

// Execute the pathfinding from start to end
List<Location> path = aStar.findPath(policeId, new Location(startX, startY), new Location(endX, endY));
```

- Then the processPath of the CityEnvironment class is called in order to move the police agents' icons in the grid:

```
CityEnvironment.getInstance().processPath(policeId, path);
```

- Lastly the path is converted to a Jason ListTerm:

```
ListTerm pathList = new ListTermImpl();
for (Location loc : path) {
    Term[] locationTerms = new Term[2];
    locationTerms[0] = new NumberTermImpl(loc.x);
    locationTerms[1] = new NumberTermImpl(loc.y);
    pathList.add(ASSyntax.createStructure("location", locationTerms));
}
```

6.2 Escorting.java Internal Action

This Internal Action manages the escorting logic for the police agents.
It activates the escorting state to true when a police agent apprehends a criminal. This change triggers an update to the police agent's icon.

- The Internal Action in the Jason police.asl file is called like this:

```
path.Escorting(ID, true);
path.Escorting(ID, false);
```

ID: police id

- The first thing done in the java file of the Internal Action is to save these values:

```
int policeId = (int)((NumberTerm) args[0]).solve();
boolean isEscorting = ((Atom)args[1]).getFunctor().equals("true");
```

- Then the methods `startEscorting` and `stopEscorting` methods of the CityModel class are called to set the state:

```
if (isEscorting) {
    // Start escorting
    env.getCityModel().startEscorting(policeId);
} else {
    // Stop escorting
    env.getCityModel().stopEscorting(policeId);
}
```

6.3 EnterJail.java Internal Action

This Internal Action manages the process for a police agent to enter the jail upon reaching it while escorting a criminal. Specifically, it involves removing the police agent's icon from the grid, signifying that the agent is inside the jail with the criminal.

- The Internal Action in the Jason police.asl file is called like this:

```
path.EnterJail(ID, Xj, Yj);
```

ID: police id
Xj: x coordinate of police agent when arrived at jail
Yj: y coordinate of police agent when arrived at jail

(Xj, Yj) are the coordinates of the police agent when it reaches the jail.
The real coordinates of the jail are (35, 35) but the agent can't move exactly in that position since it is a non-free position, thus (Xj, Yj) will be something like (34, 35) or (36, 35) that are neighbouring positions to the jail.

- The first thing done in the java file of the Internal Action is to save these values:

```
int policeId = (int)((NumberTerm) args[0]).solve();
int Xj = (int)((NumberTerm) args[1]).solve();
int Yj = (int)((NumberTerm) args[2]).solve();
```

- Then the position of the police agent is saved by calling the `getAgPos` method and the real coordinates of the jail (35, 35) are saved as well:

```
Location currentPoliceLoc = model.getAgPos(policeId);
int xJail = 35;
int yJail = 35;
```

- Then if the police agent's coordinates are within a 2-cell radius distance from the jail position, the agent is removed from the grid:

```
if (Math.abs(currentPoliceLoc.x - xJail) <= 2 && Math.abs(currentPoliceLoc.y - yJail) <= 2) {
    model.removePoliceAgent(policeId, currentPoliceLoc.x, currentPoliceLoc.y);
    System.out.println("Police agent: " + agentName + "removed at: (" + Xj + ", " + Yj + ")");
} else {
    // nothing to do
}
```

6.4 Arrested.java Internal Action

This Internal Action manages the logic to arrest a criminal which correspond to remove the criminal icon when it is found by a police agent meaning that the criminal has been arrested.

- The Internal Action in the Jason criminal.asl file is called like this:

```
path.Arrested(ID, Xc, Yc);
```

ID: criminal id
Xc: x coordinate of the criminal
Yc: y coordinate of the criminal

- The first thing done in the java file of the Internal Action is to save these values:

```
int criminalId = (int)((NumberTerm) args[0]).solve();
int x = (int)((NumberTerm) args[1]).solve();
int y = (int)((NumberTerm) args[2]).solve();;
```

- Then the method `arrestCriminal` of the CityModel class is called to remove the criminal from the grid:

```
env.getCityModel().arrestCriminal(criminalId, x, y);
```

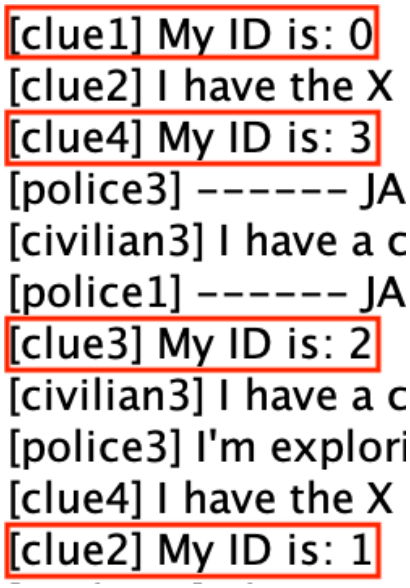
7. Utility and Helper Classes

In the project there is a “helper” folder which contains some classes specifically designed to support different aspects of the system. These utility classes provide some functions that facilitate the main operations of the program such as mapping police agents’ ids from Java to Jason or adding percpets to agents.

7.1 AgentIdMapper.java Utility Class

This utility class is fundamental to map the IDs of agents from Java to Jason.
This mapping is essential because the ID numbering systems differ between the two environments: in Java, agent IDs range from 0 to 12, whereas in Jason, the IDs span from 0 to one less than the total number of instances of that agent type.
For example, the four clue agents in Java have the IDs: 5, 6, 7, 8 while their IDs in Jason are: 0, 1, 2, 3 and the same holds for all the other agents.

```
// Police agents
city_model.setPoliceAgentPos(0, 35, 34);
city_model.setPoliceAgentPos(1, 34, 35);
city_model.setPoliceAgentPos(2, 34, 34);
// Criminal agents
city_model.setCriminalAgentPos(3, 10, 12);
city_model.setCriminalAgentPos(4, 26, 21);
// Clue agents
city_model.setClueAgentPos(5, 5, 5);
city_model.setClueAgentPos(6, 5, 35);
city_model.setClueAgentPos(7, 35, 5);
city_model.setClueAgentPos(8, 35, 30);
// Civilian agents
city_model.setCivilianAgentPos(9, 9, 4);
city_model.setCivilianAgentPos(10, 10, 35);
city_model.setCivilianAgentPos(11, 31, 9);
city_model.setCivilianAgentPos(12, 29, 29);
```



7.2 AgentPercept.java Utility Class

This utility class is used in the project to add and update the beliefs of the agents, so it acts as the intermediary layer between the agent representation in Java and Jason.
In this class a lot of methods exist, each tailored to introduce a specific type of percept to a particular agent. This functionality is enabled by the method:
`addPercept(String agName, Literal percept)` from Jason [7]

The UML diagram associated with this class provides a comprehensive overview of all the methods available:

This is a short description of them following the order as they appear in the code:

- `updateAgentPercepts`(…): method to add the percepts about the position and id of the agents.
- `addPolicePercept`(…): method to add the percepts about the police agents' start position - end position and jail position. Called by the setPoliceAgentPos() method in the CityModel class.
- `addCivlianPercept`(…): method to add the percepts to civilian agents about the position of the clue agents. Called by the setCivilianAgentPos method in the CityModel class.
- `addCluePerceptX`(…) and `addCluePerceptY(…)`: methods to add the percepts to clue agents about the position of the X or Y coordinate of criminal agents. Called by the setClueAgentPos method in the CityModel class.
- `foundAgent`(…): method to add the percepts to police agents about the position of the neighboring agents found while exploring.
- `jailWithCriminal`(…): method to add the percept to police agents about the fact that they have reached the jail with a criminal.
- `destroyAgent`(…): method to destroy the police agent when at jail with a criminal.
- `destroyAllAgents`(…): method to destroy all agents. Called by the destroyAgent method (described before) when the two criminals have been arrested.

7.3 LookAround.java Utility Class

This utility class is used from police agents at each step in order to detect the presence of other agents in the neighbouring cells of the current position.
Inside this class the only method implemented is called "checkSurroundings", it takes in

input the city model instance, the x and y coordinates of the police agent of its current position and the ID of the police agent.

- The method begins by defining two arrays, dx and dy, which represent the relative positions an agent can check around its current location in the grid. These arrays are structured to include the positions immediately adjacent to the agent, in all directions - left, right, up, down, and the four diagonals.

```
int[] dx = {-1, 0, 1};
int[] dy = {-1, 0, 1};
```

- Using nested loops, the method iterates over these two arrays to compute new x and y coordinates (nx and ny) surrounding the agent's current position (x, y). For each combination of dx and dy, the new coordinates are calculated. The loop specifically skips the iteration where both dx and dy are zero because this combination would point to the agent's current position, which does not need to be checked again.

```
for (int i = 0; i < dx.length; i++) {
    for (int j = 0; j < dy.length; j++) {
        int nx = x + dx[i];
        int ny = y + dy[j];
        // Skip the current cell
        if (dx[i] == 0 && dy[j] == 0) {
            continue;
        }
```

- Once the new coordinates are determined, the method verifies that these coordinates fall within the grid's limits using `cityModel.isInGrid(nx,ny)`. Within the valid grid boundaries, the method then checks for the presence of various types of agents or significant objects. Each time an entity is found, its type is added to the foundAgents list, which serves as a record of all entities detected around the police agent. Moreover, every time an agent or the jail is found the method foundAgents from the AgentPercept class is called to add this percept to the agents.

```
if (cityModel.isInGrid(nx, ny)) {
    if (cityModel.hasObject(CityModel.CLUE_AGENT, nx, ny)) {
        foundAgents.add("Clue Agent");
        int agId = cityModel.getAgentId(CityModel.CLUE_AGENT, nx, ny);
        AgentPercept.foundAgent(CityEnvironment.getInstance(), policeId, agId, x, y);
    }
    if (cityModel.hasObject(CityModel.CIVILIAN_AGENT, nx, ny)) {
        foundAgents.add("Civilian Agent");
        int agId = cityModel.getAgentId(CityModel.CIVILIAN_AGENT, nx, ny);
        AgentPercept.foundAgent(CityEnvironment.getInstance(), policeId, agId, x, y);
    }
    if (cityModel.hasObject(CityModel.CRIMINAL_AGENT, nx, ny)) {
        foundAgents.add("Criminal Agent");
        int agId = cityModel.getAgentId(CityModel.CRIMINAL_AGENT, nx, ny);
        AgentPercept.foundAgent(CityEnvironment.getInstance(), policeId, agId, x, y);
    }
    if (cityModel.hasObject(CityModel.POLICE_AGENT, nx, ny)) {
        foundAgents.add("Police Agent");
    }
    if(cityModel.hasObject(CityModel.JAIL, nx, ny)){
        foundAgents.add("FOUND THE JAIL");
        // If the police agent is escorting a criminal agent, jail them.
        // The icon of the police agent will return the policeImage icon since the criminal agent is jailed.
        boolean escorting = cityModel.isEscorting(policeId);
        if (escorting) {
            AgentPercept.jailWithCriminal(CityEnvironment.getInstance(), policeId, x, y);
        }
    }
}
```

8. Possible future improvements

The proposal of this project has been evaluated as “hard”, indeed initially I encountered some difficulties in understanding how to integrate Java and Jason, how to manage beliefs and associate them to the appropriate agent. Additionally implementing the movement of police agents within the grid with the update of the icon position was quite challenging.

I think that in future some improvements can be made to enhance the complexity and the functionalities of the project, for example:

- Add more agents in the grid: add some criminals, civilians, and clues.
- Make the criminal escape from the police agent.
- Improve the communication between police agents such that they can exchange clues about criminals.
- Implement more than one city model such that the simulation can be done in different environments with different obstacles, jail, and agents’ positions.

References

[1] “Jason: An Interpreter for AgentSpek”: https://jason-lang.github.io/#about-jason

[2] Jason.environment.Environment: https://jason-lang.github.io/api/jason/environment/Environment.html

[3] GridWorldModel: https://jason-lang.github.io/api/jason/environment/grid/GridWorldModel.html

[4] GridWorldView: https://jason-lang.github.io/api/jason/environment/grid/GridWorldView.html

[5] A* algorithm: https://en.wikipedia.org/wiki/A*_search_algorithm

[6] DefaultInternalAction: https://jason-lang.github.io/api/jason/asSemantics/DefaultInternalAction.html

[7] addPercept:
https://jason- lang.github.io/api/jason/environment/Environment.html#addPercept(jason.asSyntax.Literal...)