Multi-Agent Exploration in a Simulated Environment

Multi-Agent System

Paolo Leo
University of Bologna
Master in Artificial Intelligence
paolo.leo3@studio.unibo.it

July 2025

Abstract

This project simulates a research team composed of different types of robotic agents that communicate to accelerate the search process. The agents are developed using Jason, a platform based on the BDI (Belief-Desire-Intention) architecture, which is well-suited to modeling autonomous and goal-driven behaviors.

The environment is a 2D map populated with various elements, including obstacles, walkable terrain, and several Points of Interest (POIs). The application supports two operational modes: full exploration of the environment, or targeted search for a randomly placed item within a POI.

Agents are divided into two main types: a 'ground' unit responsible for interacting with the environment and a 'drone' unit capable of faster exploration. The team must coordinate effectively to explore the map and, when applicable, locate the target as efficiently as possible.

1 Introduction

Multi-Agent Systems (MAS) offer a powerful paradigm for addressing problems that require distributed reasoning, coordination, and adaptability. In many real-world scenarios, the use of collaborative autonomous agents could provide significant advantages; deploying multiple intelligent agents that collaborate, adapt, and solve complex problems in real-time, gathering and sharing information dynamically, could make the difference between a task efficiently completed or totally failed.

This project presents an implementation of a simulated 2-D environment, where a team of robotic agents has the task of efficiently explore the space to either gather useful information or locate a hidden target. To add complexity, different types of obstacles that affect agent's movement are added; the goal is to make the actors navigate through, using and sharing the locally acquired knowledge.

The effectiveness of the exploration is measured using mainly three metrics:

1. **Task Time**: the total time required for the agents to fully explore the environment and return to their starting position.
2. **Exploration Time**: the time required for the agents to investigate all the POIs.
3. **Redundant POI Interactions**: the number of times agents interact with Points of Interest (POIs) that have already been explored, which serves as an indicator of how well information is being shared.

This report presents the design, implementation, and evaluation of the system. It begins with an overview of the technologies used, followed by the system architecture, implementation details, experimental results, and concluding remarks.

2 Employed Technologies

This section contains a brief overview of the employed technologies.

2.1 AgentSpeak/Jason

According to [1], AgentSpeak is an agent-oriented programming language inspired by and based on a model of human behavior called Belief-Desire-Intention (BDI) model. An agent bases its reasoning on three mental structures:

- **Beliefs**: information the agent has about the world.
- **Desires**: all the possible states of affairs that the agent might like to accomplish.
- **Intentions**: states of affairs that the agent has decided to work towards.

Jason is an open-source interpreter for AgentSpeak that facilitates the implementation of BDI agents and multi-agent systems. Jason supports communication, plan execution, and environment interaction, making it well-suited for the purposes of this project.

2.2 Java

The project is implemented in Java, a widely used platform-independent programming language. Java was chosen since its rich ecosystem facilitates the environment development and enables straightforward integration with the Jason platform. The environment, user interface, control logic and some of the more complex agent actions are developed in Java, allowing seamless coordination with the AgentSpeak agents.

2.3 Gradle

Gradle is a build automation tool that manages tasks like compilation, packaging, testing, deployment, and publishing. It works well with various programming languages, such as Java, Kotlin, Scala, Groovy, C++, and Swift. It simplifies the build process by automating repetitive tasks and ensuring consistency across different environments.

In this project, Gradle is used to manage the Java build process and handle dependencies (such as Jason), making the system easier to configure, run, and reproduce.

2.4 D*Lite

The D* Lite algorithm, [2], was developed by Sven Koenig and Maxim Likhachev for a faster and more lightweight alternative to the D* algorithm.

It is an incremental search algorithm used for path planning problems in partially known or dynamic environments that, despite its reduced computational complexity, manages to keep its optimality. Unlike other algorithms of its kind, it works backwards, from the goal to the start and, similar to Dynamic A*, it can account for changes in the graph, without recalculating the entire graph from scratch. It uses a priority queue to manage state updates efficiently and is particularly well-suited for autonomous agents operating in environments where new information may be revealed during navigation.

For these reasons, it was chosen for this project.

3 System Architecture

This section contains a description of the system architecture, focusing on the main components and explaining how they work and interact to fulfill the task.

3.1 Project Folder Structure

As illustrated in Figure 1, the project it is wrapped using gradle and it is structured as follows:

- **app/src/main/java**: Contains the core source code, divided into packages:
  - `ag/`: Implements agent classes:
    - `MyAgent.java`: Superclass for all agents.

    - `GroundAgent.java`, `AirAgent.java`: Specialized subclasses for ground and air agents.
  - `env/`: Handles the creation, rendering, and management of the environment, in particular:
    - `envModel.java`: Maintains environment state and performs actions.
    - `envView.java`: Graphically shows and update the environment.
    - `envController.java`: Manages the communication among model, view and agents.
    - `StartupConfig.java`: Contains the initial parameters setup.
  - `tile/`: Manages the Tiles used in the graphics:
    - `Tile.java`: Tile class that contains basic tile information.
    - `TileManager.java`: Loads and provides tile data and map info.
  - `utils/d_star_lite/`: Implements D* Lite algorithm.
  - `utils/internal_actions/`: Custom internal actions performed by agents:
    - `air/`, `ground/`, `shared/`: Actions are organized by agent type or shared logic.
  - `utils.TermUtils.java`: Implements various methods used for working with Terms.
- **app/src/main/resources**: Stores resource files:
  - `maps/`: Map layout files.
  - `tiles/`: Images for tile rendering.
  - `air.asl`, `ground.asl`: AgentSpeak files for air and ground agents.
- **robots.mas2j**: MAS configuration file.
- **build.gradle** and **settings.gradle**: Gradle scripts to manage dependencies and build logic.
- **gradlew** and **gradlew.bat**: Gradle wrapper scripts for Unix and Windows environments.

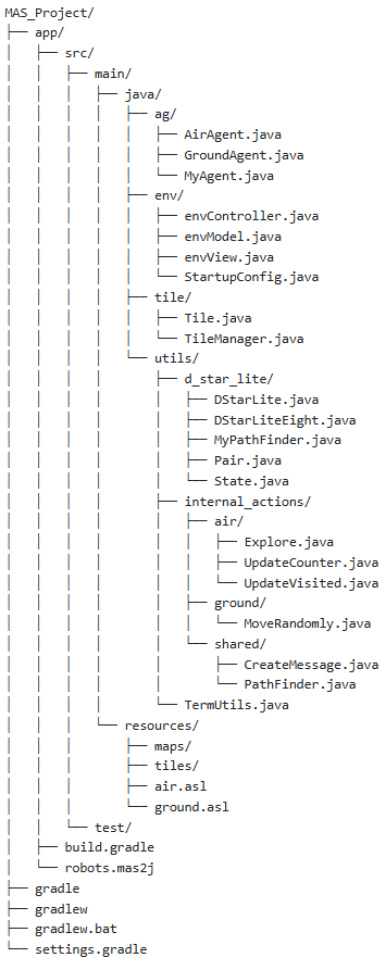

Figure 1: A tree view of the Project Folder.

3.2 Architecture Paradigm

For structuring the system logic, a Model-View-Controller pattern (MVC) has been used. Figure 2 represents, using an UML-like diagram, how all the components of the system interact each other.

The interaction begins when the user starts the application, selecting the initial configuration. This triggers the Controller, which first instantiates the Model and View, and then acts as a coordinator between them.

Moreover, the controller manages the communication between the Model and the agents, delivering the action to be performed to the first, and the resulting percepts to the second. The Model maintains the current state of the environment and informs the View about what needs to be displayed; the view manages all the graphical updates based on the model current state.

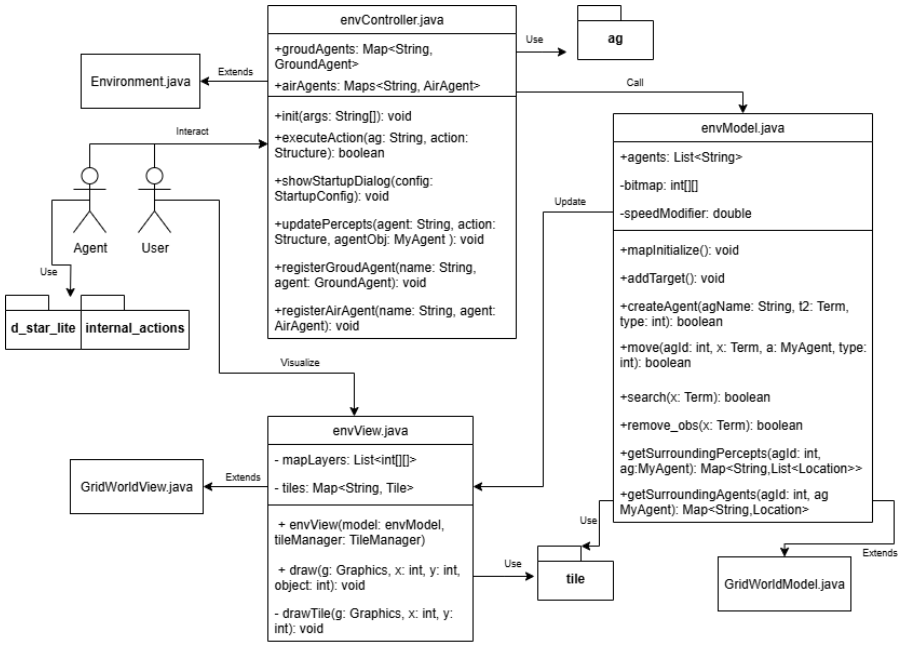

Figure 2: An UML-like diagram to show the class interactions.

3.3 Principal Components

This section provides an overview of the implementation details for the different modules involved in the project.

3.3.1 Ground Agent

Ground Agents are the only entities that can interact with the environment: they can remove obstacles and investigate Point of Interests (POIs). Their behavior depends on the operational mode selected by the user (either exploring all POIs on the map or locating a specific target hidden within one of them).

These agents are slow units, they can walk at normal speed on streets (1.5 seconds per step) and at reduced speed on grass. Their movement and perception are limited to the four cardinal directions, with a range of one cell.

Communication between Ground Agents is only possible when they can perceive each other, i.e., when they are located on adjacent cells. In contrast, communication with the Air Agent does not require direct perception, since it is possible that the aerial unit may initiate the interaction and request information.

Figure 3 shows the logic used by the Ground Agent to decide what to do, considering also the information it has available. It is important to know that:

1. Agents mark as '*non-visited*' all the walkable terrain that they know exist but they have not visited.

2. Before the task starts, a priority index is incrementally assigned to each of them. It is used for coordination.

3. They use the *D*Lite* to find all the paths. The algorithm is recomputed at each step to find the shortest route and it has been slightly modified to work in four directions.

A Ground Agent follows four primary behavioral steps:

- **Exploring Cycle**: A Ground Agent can continue exploring as long as it has an adjacent *non-visited* cell; the choice of the cell is random. After each step, it always check whether there is an available *goal* (a POI which the agent believes has not yet been investigated).
- **Getting Unstuck**: A Ground Agent is *stuck* if there are no adjacent *non-visited* cells. This may occur in dead ends, or after fully exploring an area in a circular pattern. When stuck, the agent computes the path to the nearest *non-visited* cell and follows it step by step. After each step, the agent checks whether a new *goal* has become available. If no goal is found after completing the path, the agent starts exploring again.
- **Going for a POI**: If a POI is found (or it has been communicated), the agent attempts to reach it. If it is aware that another agent is heading toward the same goal, the priority index determines which agent should proceed. This prevents multiple agents from pursuing the same goal in vain. The agent that drops the goal will check if there are alternatives before returning to explore. Even here, if a closer goal is discovered, the agent will prioritize it.
- **Returning home**: If a target is found, agents will return home directly (since the task has been completed). Otherwise, they return home when they can no longer explore (there are no more known *non-visited* cells) and all the known POIs have been investigated. In this second case, while returning, they may still deviate to investigate a POI if it becomes available.

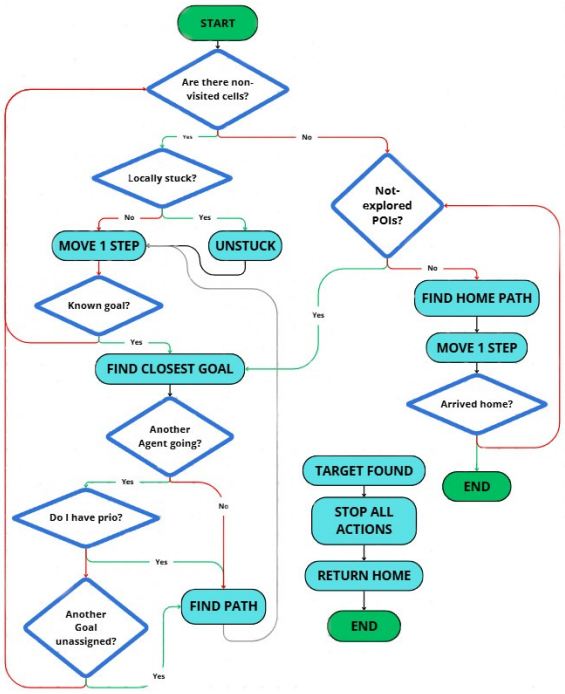

Figure 3: The Ground Agent logic schema.

3.3.2 Air Agent

Air Agents are fast units whose sole purpose is to explore the environment quickly and inform the ground agents about the structure of the map. They move almost three times faster than Ground Agents and in all directions. At each timestep, they perceive an entire 5×5 area around them. They cannot interact with the environment in any way and the only obstacles they must avoid are map contours and mountains.

Due to their broader perception range, Air Agents typically initiate communication with Ground Agents, prompting them to respond and share information.

Overall, they are much simpler agents. Things to know before diving deep into their behavior are:

1. They mark as '*non-visited*' all the walkable terrain that they know exist but they have not visited.
2. Unlike Ground Agents, they mark as '*visited*' not only the cell they visit, but also the 3×3 area around it.
3. They use the *D*Lite* to find all the paths.
4. To balance between exploration and communication, they use a counter, let's call it *commTrigger*, incremented whenever new information is discovered and decremented after knowledge has been shared with another agent.

Figure 4 outlines how also their actions can be divided into four distinct steps:

- **Exploring-Unstuck Cycle**: The logic is similar to Ground Agents; they distinguish between *visited* and *non-visited* cells, and continue exploring as long as there are adjacent *non-visited* cells available. The selection strategy is locally greedy: they choose the adjacent *non-visited* cell that is surrounded by the highest number of other *non-visited* cells.

  They can end up stuck locally and, like Ground Agents, they use *D*Lite* to reach the closest *non-visited* cell.

- **Search an Agent**: When *commTrigger* reaches a certain threshold, the Air Agent attempts to find a Ground Agent to share the newly gathered information. It maintains, as a belief, the last known position of each Ground Agent and selects a different one as a target each time communication is triggered.

- **Loop Around and Returning home**: Once all map cells have been explored, the Air Agent begins the final coordination phase to ensure all Ground Agents receive the full map information. Since Ground Agents tend to move between POIs, the Air Agent alternates its visits between POIs and the starting location in order to increase the chances of meeting them. Only after all Ground Agents have returned, the Air Agent also returns to complete its task.

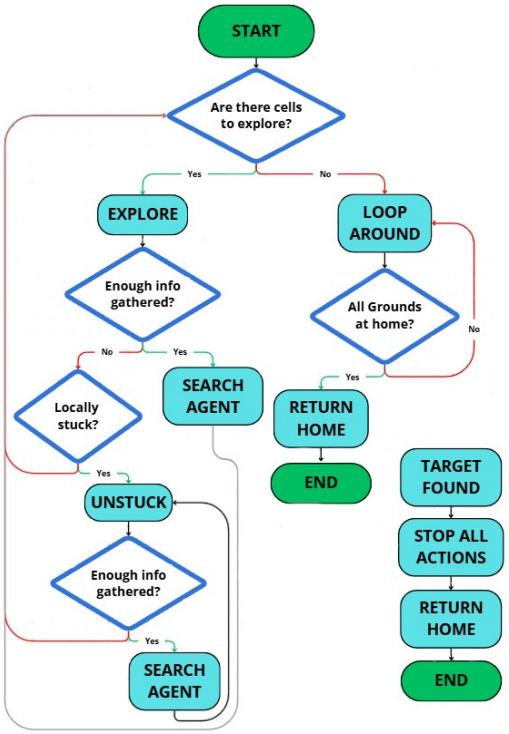

Figure 4: The Air Agent logic schema.

3.3.3 Environment

The environment is a 2-D tile based grid.

Basic functionalities, such as grid structure, cell management or agent placement, are already provided by Jason through the *GridWorldModel* and *GridWorldView* classes.

The graphical tiles used for rendering the environment were obtained from a free and open-source asset pack available on 'Itch.io', [3], and are used in accordance with the license provided by the creator. Minor adjustments were made to better align the visuals with the specific needs of the project.

A map is defined by two text files:

1. **tilemap.txt**: The Tile Map contains, layer by layer, the unique id of each tile in each position. Multiple layers are necessary since in each cell it is possible to have multiple tiles. It is used by the View to show the right tiles.

2. **bitmap.txt**: Contains two sections: the first is the correspondence between the values used later in the file and the mask values used by the model, the second is a grid of dimensions $mapLength \times mapHeight$, where each cell corresponds to one of the major objects (described below). It is used by the model to instantiate the appropriate object in each cell.

The environment is composed by the following items:

- **Edges and Mountains**: Define the contours of the map and cannot be traversed by either Ground Agents or Air Agents.

- **Street**: Main walkable terrain. Ground Agents can walk on them without any cost at normal speed. Does not affect Air Agents.

- **Grass**: Difficult terrain. Ground Agents can walk on them with reduced speed; therefore they have an higher cost and are avoided if possible. Does not affect Air Agents. In the map, green and red ground is considered grass.

- **Removable Obstacles**: These block the path of Ground Agents but can be cleared at a cost of 4 seconds each. Ground Agents must decide whether to remove them or, if possible, navigate around them to save time. They do not affect Air Agents.

- **Non-Removable Obstacles**: Objects that block the path of Ground Agents but cannot be removed. They must be bypassed. They do not affect Air Agents.

- **Points Of Interest (POIs)**: Only Ground Agents can interact with POIs. When operating in "search the Target" mode, the target is instantiated within one of them; note that the target can be found only interacting with the POI. Visually, POIs are outlined in red if they have not yet been interacted with, and in green once they have. A yellow circle around a POI indicates the presence of the target.

Figure 5: A snapshot of the map in which is possible to recognize all the elements.

3.4 Internal actions

Jason gives the possibility to create internal actions, i.e., built-in operations that agents can perform in addition to their own plans and beliefs. These actions extend the agent's reasoning capabilities, making it perform more complex task.

Each internal action is defined as a Java class and it must extend *DefaultInternalAction*, the built-in implementation of the internal action interface.

In the project, they are organized in three folder, depending on who is going to use them. This sections comprehends a brief overview of the internal actions defined in the project and the reasoning behind them.

3.4.1 Explore.java

It is used by the Air Agent to explore the environment. As previously mentioned, it follows a locally greedy strategy: at each step, the agent selects its next move randomly among the valid, adjacent cells (to him, in all 8 directions) that have the highest number of valid, adjacent non-visited cells.

This type of exploration may not be optimal in the long run, as it can lead to inefficient coverage patterns. However, it tends to maximize the discovery of new areas in the short term, which is particularly beneficial in the early stages of the task, when we want to find a POI as soon as possible to guide the Ground Agents toward meaningful regions of the map.

3.4.2 UpdateCounter.java

It is used by the Air Agent to update the *Exploration Counter* (section 3.3.2). New information increase the counter and searching an agent to communicate them, decreases it. Cells that already affected the counter are marked as `processed(X,Y)` in the agent's Belief Base. `X` and `Y` are the cell coordinates.

The algorithm takes as input the list of the processed cell and returns to the agent the value to add to the counter. Each cell type affects the counter differently, the more important it is, the higher is the value added.

3.4.3 UpdateVisited.java

It is used by the Air Agent to mark as `visited(X,Y)` its current position and all the eight cells around it. It is called each time the drone moves.

3.4.4 MoveRandomly.java

It is used by Ground Agents to explore. The function takes as input the lists of known streets and already visited cells, and randomly selects a valid street to move to. For Ground Agents, a valid street is defined as a non-visited street cell adjacent in a cardinal direction (north, south, east, or west).

Ground Agents exploration happens only on streets (section 3.6).

3.4.5 CreateMessage.java

Is used by both agents type to communicate. It takes as input a list of functors to share and a list of already sent beliefs. A belief from the agent's Belief Base is included in the message if:

1. its functor is present in the list of functors to communicate;
2. it has not already been sent to the target agent.

Each time a list of beliefs is sent to another agent, every belief in that list is marked as `already_sent(AGENT, BELIEF)`, where `AGENT` is the receiver agent. This mechanism is essential to avoid redundant communication and to prevent exponential growth in the number of beliefs being sent during each iteration as time progresses, which would otherwise slow the system down significantly. In this way, retrieve already sent messages, for each receiver agent, is trivial: `.findall(X, already_sent(NAME, X), SENT)`.

3.4.6 PathFinder.java

This action is used by both types of agents to compute the optimal path from their current position to a goal, based on their local knowledge of the environment. If multiple goal positions are provided, the algorithm selects the closest one according to the chosen heuristic. It returns both the selected goal and the path to reach it. The inputs to the algorithm are:

- The current position;
- The list of objects with cost different from 1; i.e., non-removable obstacle have -1 cost, grass has cost 2, removable obstacles have cost 4.
- The list of goals.

Due to the differences in movement capabilities between Ground and Air Agents, two distinct instances of the D* Lite algorithm are implemented:

- Ground Agents can move only in the four cardinal directions, so the algorithm uses the Manhattan distance as the heuristic.
- Air Agents can move in all eight directions, so the algorithm uses the Octile distance heuristic, which better reflects diagonal movement costs.

The two implementation also differ in how they compute their set of *Successors* and *Predecessors* for each node; they have to reflect the movement directions. They can be found in the folder `app/src/main/java/utils/d_star_lite`.

3.5 Starting Menu

A menu is displayed before launching the application. From there, the user can choose the initial settings, such as the number of Removable Obstacles, the simulation speed, the goal type, and the map to be used.

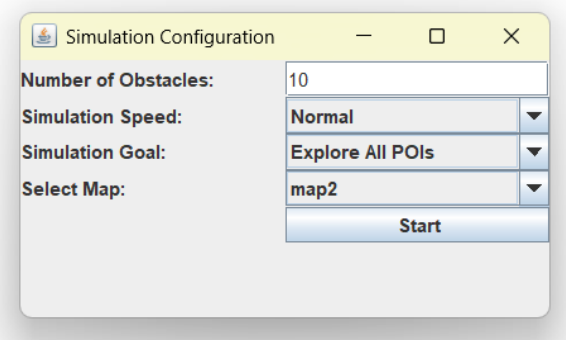

Figure 6: The starting menu.

3.6 Observations, Limits and Assumptions

- The system is designed to use only one Air Agent. While it is technically possible to deploy two, communication between them has not been implemented. This design decision is mainly due to the limited map size. Using more than one Air Agent would result in the map being fully explored before the Ground Agents have made significant progress. Additionally, information sharing would become too easy, as the map would be over-covered, reducing excessively the challenge of the task.
- Ground Agents can explore only by moving along streets. This implies that, for them to complete a full exploration independently (i.e., without the help of drones), all POIs must be adjacent to a street, and all streets must be interconnected. This limitation does not apply when a drone is used, since isolated POIs can still be discovered from the air and communicated to the Ground Agents. As long as these POIs are reachable, the agents can navigate to them. This constraint can be deliberately leveraged to increase the complexity of the exploration task by placing POIs in hard-to-reach locations.

- Another important factor to consider is the discrete settings provided by Jason that can affect Ground Agents communication.
  Communication occurs when an agent perceive another in an adjacent cell, and percept updates are processed at the starting of the each reasoning cycle. However, due to timing misalignment, there are rare cases in which communication fails even when agents are correctly positioned next to each other. In some instances, this results in one-sided or entirely missed exchanges.
  This issue has been partially addressed by forcing Ground Agents to send a response when they receive a message. Despite this improvement, a few instances of miscommunication were still observed during testing.

- As previously mentioned, drones use a counter and a threshold to balance between exploration and communication. However, the current logic used to update the counter has a significant flaw in its reduction phase.
  The counter is decreased when the drone reaches the last known position of the agent it is looking for, not when communication actually occurs. While this approach helps prevent the Air Agent from wasting too much time searching for an agent and favors exploration, it has an important downside: it can result in losing track of agents entirely, especially when only a few of them are employed. This can lead to suboptimal outcomes.

4 Simulation Results

As previously mentioned in the introduction, the system will be evaluated using three metrics:

1. **Task Time (TT)**: the total time required for the agents to fully explore the environment and return to their starting position.

2. **Exploration Time (ET)**: the time required for the agents to investigate all the POIs.

3. **Redundant POI Interactions (RPI)**: the number of times agents interact with Points of Interest (POIs) that have already been explored, which serves as an indicator of how well information is being shared.

An important relationship between Task Time and Exploration Time is their difference, call it Post-Exploration Delay:

$$\text{PED} = \text{Task Time} - \text{Exploration Time} \tag{1}$$

which represents the delay between the primary task completion (exploring all the POIs) and the final termination (returning home). A large gap suggests that agents may be spending unnecessary time navigating the environment after all POIs have already been investigated. This may happen due to inefficient communication or coordination.

In parallel, the Reduntant POI Interactions metric also gives insights on how effective communication is. When communication is weak, agents may revisit POIs that have already been checked, wasting time and effort. However, this metric is inherently dependent on the number of Ground Agents. With more agents, the potential for redundancy naturally increases.

Considering that the map used for testing has 13 POIs, in fact, the maximum number of Redundant Interactions is given by:

$$\text{maxI} = (\text{numGroundAgents} - 1) \times 13 \tag{2}$$

A normalized version can also be defined to facilitate comparison across experiments with different number of agents, let's call it Normalized Redundancy, NR:

$$\text{NR} = \frac{\text{Redundant Interactions}}{maxI} \tag{3}$$

The Operational Mode used for the tests is *Explore All POIs*, as it is more consistent and not affected by the random target placement. Each configuration will be tested using different combinations of agents, with 5 trials per setup, *Normal Speed* and *15* initial obstacles.

For each configuration, a table will contain all the results. Each table will show the Task Time (TT), the Exploration Time (ET), the Post-Exploration Delay (PED), the number of Redundant POIs Interaction (RPI) and the Normalized Redundancy (NR), for each try. A final table will

contain, for each combination, an average of all the partial results.

Table 1 shows the results when using only three Ground Agents. Despite the relatively high number of active agents, the performance is quite poor. The average Task Time exceeds 6 minutes per run, and the number of redundant POI interactions is around 11. The Exploration Time is quite low, meaning that the Post-Exploration Delay is very large. The two metrics, in fact, are almost equal, which is not a good sign.

The high completion time is mainly due to the need, for each agent, to individually explore the map and the lack of efficient communication. Redundant searches occur frequently because communication is sporadic, agents can only exchange information when they happen to encounter each other. In other words, even when map has completely been explored, agents will still complete their search due to lack of communication.

Table 1: 3 Grounds, 0 Air.

Try Number	TT	ET	PED	RPI	NR
Try 1	06:27	03:23	03:04	12	0.46
Try 2	05:45	02:58	02:47	11	0.42
Try 3	07:33	02:57	04:36	11	0.42
Try 4	06:16	02:51	03:25	11	0.42
Try 5	05:11	03:40	01:31	8	0.30

Table 2 shows that even when the number of Ground Agents is reduced, the addition of an Air Agent significantly improves performances. The average Task Time decreases by about two minutes but, what really makes the difference, is the number of redundant searches, together with the PED.

The PED is particularly noteworthy: it averages only 34 seconds, meaning that once all the POIs have been explored, the agents, almost straightforwardly, start returning to the home base. Considering also that the Exploration Time has been slightly reduced, it is clear that the introduction of an Air Agent improved significantly the communication dynamics.

Furthermore, RPI drops dramatically, from an average of 12 to just 1 per run. The Normalized Redundancy, which adjusts RPI according to the number of Ground Agents, has reduced drastically. This happens because information is well spread across the map: the drone communicates both what already has been done, and which areas must be explored. This makes the exploration way more efficient.

Table 2: 2 Grounds, 1 Air.

Try Number	TT	ET	PED	RPI	NR
Try 1	04:30	03:36	00:54	5	0.38
Try 2	04:06	03:32	00:34	0	0.00
Try 3	04:31	03:49	00:42	0	0.00
Try 4	03:46	03:22	00:24	1	0.07
Try 5	04:02	03:45	00:17	0	0.00

Finally, table 3 shows how both Task Time and Exploration Time can be further reduced by adding more Ground Agents. With more agents, in fact, more areas or POIs can be explored simultaneously, and information spreads more easily across the map due to its limited size. Compared with the previous experiment, TT is reduced by nearly 10% and the ET by approximately 20%, indicating a significantly faster navigation and task execution.

Theoretically, the drawback of employing more Ground Agents is to have more redundancy. However, as the Post-Exploration Delay and Normalized Redundancy show, the increase in this case is negligible. Specifically, PED increases by only 13 seconds, and NR rises just by 0.02 points.

The presence of the Air Agent ensures that Ground Agents remain informed about the overall map status, this effectively minimizes the number of redundant POI interactions and maintains efficiency even as agent count increases.

Table 3: 3 Grounds, 1 Air.

Try Number	TT	ET	PED	RPI	NR
Try 1	03:34	02:45	00:49	0	0.00
Try 2	03:38	03:10	00:28	6	0.23
Try 3	04:10	03:16	00:54	3	0.11
Try 4	03:59	03:13	00:46	3	0.11
Try 5	03:30	02:31	00:59	3	0.11

Table 4: Average results table. Values (except NR) are rounded to the closest integer

Agent Configuration	TT	ET	PED	RPI	NR
3 Grounds 0 Air	06:14	03:10	03:05	11	0.40
2 Grounds 1 Air	04:11	03:37	00:34	1	0.09
3 Grounds 1 Air	03:46	02:59	00:47	3	0.11

Conducting additional tests with a higher number of agents would not provide meaningful results due to the limited size of the map. Introducing more Air Agents would make the exploration phase disproportionately fast, without offering valuable insights. Similarly, increasing the number of Ground Agents could either lead to overcrowding (when multiple agents redundantly exploring the same areas) or to a reduced exploration time purely due to their quantity, rather than to an efficient coordination and communication.
Similarly, testing with fewer agents would likely produce worse outcomes that follow the same behavioral patterns already observed in previous scenarios.

5 Reproducibility

To reproduce the system, download the code from `https://github.com/PaoloLeo99/MAS_Project_Leo.git` and extract it in a folder. From command line navigate to the directory where the project is:

```
cd my-project/
```

Then to start the task, on Windows:

```
gradlew.bat runExplorationTask
```

In macOS / Linux:

```
./gradlew runExplorationTask
```

Note that *Java 21* or a more recent version needs to be installed. To select the number of agents, the *robots.mas2j* must be modified.

6 Conclusion

This project designed and implemented a multi-agent system for exploring 2D grid-based environment using two type of agents; i.e a Ground Agent, capable of interacting with Points of Interest (POIs), and Air Agents, for rapid exploration and information spreading.
The system demonstrates how the presence of a drone unit can dramatically enhance overall efficiency. Indeed, it is possible to complete the task using only multiple Ground Agents but, their exploration is slow and, due to limited communication, POI investigation becomes redundant. The introduction of a single Air Agent significantly improves information flow and coordination, leading to faster task completion and reduced redundancy.

Future work may include:

- Ground and Air agent logic could be further enhanced. This can be done refining the communication protocol to improve the relevance and quality of the shared information, as well as enabling agents to use that information more intelligently. Another solution could be implementing some Machine Learning technique to improve agent coordination and movement. Multi-Agent Reinforcement Learning would fit the problem really well.

- Make the environment more dynamic and interactive, for example, adding timed events or sub tasks necessary to complete in order to progress. These additions would increase the complexity of decision-making and require a more refined planning and coordination.
- Make the task harder constraining agents capabilities. For example, agents could have limited energy, requiring them to recharge periodically, or be subject to malfunctions. These constraints would require the system to be more robust.
- Having larger maps would require more agents for an efficient exploration. In such settings, implementing communication and coordination mechanisms among Air Agents becomes essential to ensure a good map coverage. Additionally, introducing new agent types with specialized roles would be interesting to enhance the task distribution and analyze the results using different team compositions.

References

[1] Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason*. Chichester, UK: John Wiley & Sons, 2007.

[2] Sven Koenig and Maxim Likhachev. "D*Lite." In: Jan. 2002, pp. 476–483.

[3] Pipoya. *RPG World Tileset*. Available at `https://pipoya.itch.io/pipoya-free-rpg-world-tileset-32x32-40x40-48x48`.