

Agent Speak Pommerman

Enrico Morselli - 979685
`enrico.morselli@studio.unibo.it`

Sara Vorabbi - 1026226
`sara.vorabbi@studio.unibo.it`

The aim of this project is to implement Pommerman, a modified version of Nintendo's classic game Bomberman (1983) [7]. Pommerman features four opposing agents that compete in a free-for-all (FFA) battle. Each agent's objective is to strategically place bombs on the board to eliminate opponents, collect power-ups, and evade the explosions from other agents' bombs, ultimately striving to be the last agent standing. The project utilizes AgentSpeak, an agent-oriented programming language, along with its interpreter, Jason, which together provide a suitable framework for developing multi-agent systems.

1 Goal/Requirements

The objective of this project is to implement a multi-agent system that simulates the environment and dynamics of Pommerman, an alternative version of the famous game Bomberman developed by Nintendo in 1983. In the original multiplayer mode, players strategically place bombs on the map to eliminate each other. These bombs explode after a set time, with a specific range of action, and can kill other players and destroy obstacles, revealing power-ups that offer specific advantages when collected. Our version of Pommerman retains almost all of these elements, with the ultimate goal for each agent being to win the game by eliminating all other agents. This implementation uses AgentSpeak, an agent-oriented programming language based on logic programming, along with Jason, an extension of AgentSpeak implemented in Java that facilitates the development of multi-agent systems.

About the use of Reinforcement Learning

The original plan for this project included the use of Reinforcement Learning (RL) to enable agents to learn from past mistakes and make better decisions during gameplay. However, implementing an RL framework in Java presented several challenges. First, we

should have learned how to use RL in Java, since our experience in the MSc program has been primarily with Python. This, nonetheless, introduce additional complexity, given the fact that we were also learning a new programming paradigm (agent-based programming) and a new programming language (AgentSpeak). Moreover, the limited availability of Java libraries for deep learning further complicated this process due to the lack of references and support. Considering these challenges, along with time constraints imposed by other projects, we decided to narrow the project scope to implementing a rule-based agent. This agent makes decisions based on a set of predefined rules without leveraging learning from past experiences. We see the integration of Multi-Agent Reinforcement Learning (MARL) as a potential area for future development.

1.1 Requirements

As outlined in the previous section, the main objective is to implement the Pommerman game. This section details the specific project requirements.

The Board

The game board is an 11x11 grid that serves as the environment where the bomber agents compete. It includes walkable paths (alternatively called *free cells* or *passable cells*), rigid walls, and wooden walls. The *rigid walls* have fixed positions on the map for each run of the simulation; they are indestructible and remain permanent throughout the game. In contrast, the *wooden walls* are randomly placed at the start of each run, occupying approximately 36% of the free cells. These walls can be destroyed by the flames of the bombs placed by the agents, which in turn creates new walkable paths. Some wooden walls, when destroyed, can release power-ups that can be picked up by the agents that receive a boost.

To prevent agents from being trapped by wooden walls at the start of the game, we ensured that the two cells adjacent to each agent's spawn point, as well as the two cells immediately beyond them, are free of wooden walls. For example, if an agent starts in the (0,0) corner, the cleared cells are: (0,1), (1,0), (0,2), (2,0). Originally, we only set the cleared cells to be (0,1) and (1,0), but we noticed that agents often died when placing a bomb if wooden walls were positioned two cells away, such as in (0,2) and (2,0). Clearing these additional cells reduces the risk of agents being caught in their own explosions, allowing them to progress further in the simulation. This configuration is managed in the `PommerModel.java` class. Figure 1 shows the environment at the beginning of the simulation.

The Agents

There are four *bomber agents* that participate in the game within the environment. After conducting experiments with different options for handling of bomb detonation, we decided to delegate this task to additional agents, called *detonators*, that manage bomb detonations behind the scenes. Our first attempt was starting a new thread every time the bomber laid a bomb on the map. This thread would handle the execution of the bomb

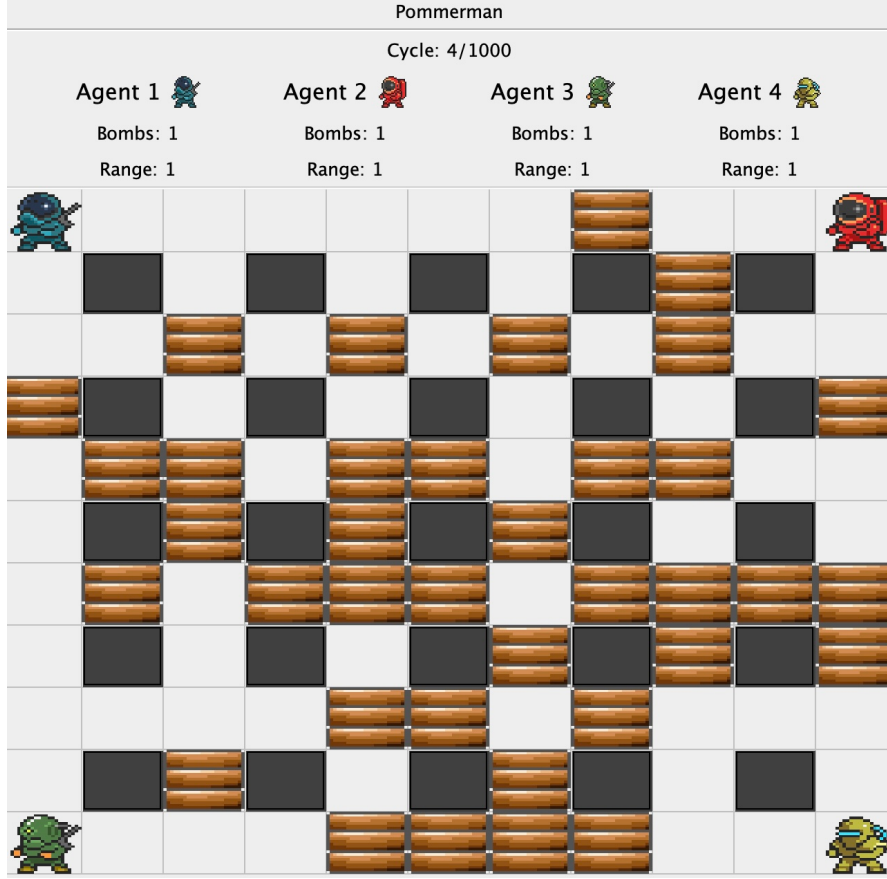


Figure 1: Environment at the beginning of the game

detonation after few seconds. Unfortunately, this caused problems in the environment simulation such as the same bomb exploding multiple times or in wrong positions on the map, and the agent not having enough time to escape after the placing of the bombs. The second attempt was done by changing the type of the Jason Environment: switching from the default `Environment` class to the `TimeSteppedEnvironment` [5] which, as the name suggests, introduces the concept of *time step*. In this type of environment every action is executed in one time step and the actions of the agents get to be “synchronised”. Another useful feature present is the possibility to delay an action by a given number of time steps. We planned to execute the *detonate* action after three time steps; In this way the bomber who placed the bomb could escape in the meantime. This also did not work as expected: Problems such as the agent dropping the bomb and standing still on that position up until the bomb detonated arose, resulting in the agent dying immediately. We could not manage to understand the exact cause of this problem, but it seemed the bomber agent waited for the execution of the *detonate* action before doing other actions such as searching a safe direction and move. That is why we decided to delegate the detonate action to another agent, the *detonator*. The detonator serves as a

kind of a bomber’s helper, it detonates the bombs whenever the bomber communicates to him that he is in a safe position. Alternatively, it can be viewed as an actual bomb detonator which the bomber can use, and waits until he knows he is in a safe position. So we added four detonator agents, each one corresponding to one bomber agent (Note: we also kept the `TimeSteppedEnvironment`). These detonator agents operate off-grid and are not visible on the game board. Details about the interaction of bomber and detonator agent will be provided later on.

We also point out that, due to time constraints and challenges encountered when developing the project, we have only implemented the Free For All (FFA) modality, where the bomber agents compete against each other.

A Pommerman game can be summarized in these four steps:

- **Starting position:** Each one of the four bomber agents start from one of the four corners of the grid, as shown in Figure 1.
- **Agent actions:** At each time step, agents can move to adjacent walkable cells (up, down, left, or right), remain stationary, or place a bomb. Initially, agents have only one bomb, and they can increase the number of bombs by picking up `inc_bombs` power-up, which are released when wooden walls are destroyed.
- **Bombs:** Bombs have an initial blast range of one square in each direction (up, down, left, and right). This range can be increased by picking up the `inc_blast` power-up released by wooden walls as for the `inc_bombs`. Bombs are detonated by the corresponding detonator agents. When a bomb explodes, any agent within its range is eliminated from the game.
- **End of the game:** The game concludes when only one agent remains alive.

The game will be implemented using AgentSpeak and Jason. The four bomber agents will operate within the environment in a manner inspired by the Pommerman Simple Agent [6], which will be detailed later.

1.2 Scenarios

The ultimate goal of this project is to replicate the game dynamics of Bomberman [7] through agents autonomously interacting with each other. Consequently, user interaction will be limited to observing the progress in the game between the four autonomous bomber agents.

The success of the project will be demonstrated by the smooth execution of the simulation, which ends when a single agent remains alive.

2 Requirements Analysis

Our Multi-Agent program is implemented using the AgentSpeak agent-programming language and its Java Extension, Jason. In the following subsection, we will briefly describe the model on which AgentSpeak is based.

2.1 Model

AgentSpeak is inspired by the *belief-desire-intention* (BDI) model [1], which is a model of human behavior. The key components of this model are:

- **Beliefs:** Information that the agent has about the world, stored in the agent's Belief Base (BB).
- **Desires:** Potential states of affairs that the agent *might* want to achieve, which can be thought of as *options* available to the agent.
- **Intentions:** The specific states of affairs that the agent is currently committed to achieving.

The decision-making model underlying the BDI paradigm is known as practical reasoning, which is *reasoning directed towards actions* — meaning — the process of deciding what to do. In simple terms, human practical reasoning consists of two main activities: *deliberation* (deciding on the states of affairs we want to achieve, i.e., our intentions) and *means-ends reasoning* (determining how to act in order to bring about these intentions).

A BDI agent that uses practical reasoning operates according to a basic control loop: (1) The agent first observes the environment to obtain the next *percept* and updates its beliefs accordingly. (2) The agent determines its desires, or options, based on its current beliefs and intentions. (3) From these options, the agent selects some to become intentions and chooses a plan to achieve these intentions. (4) The agent executes each action from the plan in sequence. (5) After executing an action, the agent pauses to observe the environment again, updating its beliefs. (6) The agent reassesses whether it should reconsider its current intentions. (7) Finally, the agent checks whether the current plan remains sound given its intentions and beliefs. If the plan is no longer deemed viable, the agent creates a new plan. The Jason interpreter follows a reasoning cycle similar to this BDI decision loop.

3 Design

The source files of the project were structured as follows:

```
src
├── agent
│   └── BomberAgent.java
├── agt
│   ├── bomber.asl
│   └── detonator.asl
├── env/example
│   ├── PommerEnv.java
│   ├── PommerModel.java
│   └── PommerView.java
├── jia
│   ├── compute_distances.java
│   ├── find_safe_direction.java
│   ├── get_direction_to_object.java
│   ├── get_random_direction.java
│   ├── is_object_in_radius.java
│   ├── is_position_unsafe.java
│   ├── is_safe_to_drop.java
│   ├── manhattan_dist.java
│   ├── my_id.java
│   └── resources/img
├── utils
│   ├── Bomb.java
│   ├── QueueElement.java
│   └── Utility.java
```

3.1 Structure

The primary entities modeled in this project are the Bomber Agent and the Detonator Agent. Their structure, including initial beliefs, goals, and plans, is defined in `bomber.asl` and `detonator.asl`, respectively. The environment is defined within the `env/example` package. To organize the environment code, we implemented the Model-View-Controller (MVC) design pattern using the `PommerEnv`, `PommerModel`, and `PommerView` Java classes.

Specifically, the `PommerModel` class serves as the Model. It defines the structure of the grid, the entities within it, and the Java methods that execute operations in the environment, such as `move()` and `dropBomb()`. The `PommerView` class functions as the View, responsible for defining the graphical interface and rendering the environment's components. Lastly, the `PommerEnv` class acts as the Controller, bridging the Model and the View. It manages the agents' actions by invoking the appropriate methods in the Model and is responsible for updating the agents' percepts whenever they move. Each

agent’s percepts are limited to a 5x5 window, which helps reduce the computational load during updates.

Another critical component of this project is the `jia` package, which contains the Jason Internal Actions. These actions are a significant feature of Jason, allowing developers to customize agent behavior. Although the Jason library includes a set of standard internal actions, it also provides the flexibility to extend these actions to implement custom behaviors. Our project extensively utilizes Custom Internal Actions by extending the `DefaultInternalAction` class and overriding the `execute()` method. This method is automatically called when the internal action is invoked and returns an object representing the action’s result.

Additionally, we implemented a custom agent class, `BomberAgent.java`, which overrides the `selectEvent()` and `selectIntention()` functions for the bomber agents. These functions are responsible for selecting the event to handle and determining the intention to pursue in the current reasoning cycle, respectively.

3.2 Behaviour

When the agent starts, it has three initial beliefs:

1. `alive` indicating that the agent is alive
2. `blast_strength(X)` representing the blast strength of the bombs it can place
3. `ammo(X)` representing the number of bombs the agent currently possesses

The blast strength and ammo are initially set to `X = 1` and can be increased by collecting power-ups on the map. In particular, the value of the `ammo` belief is decreased by one every time the bomber agent places a bomb, and the bomb explodes, and it is incremented by one when a bomb previously placed by the agent explodes. Upon starting execution, and after the `alive` belief is added to its belief base, the agent broadcasts a message (`im_alive`) to all other agents, indicating its presence in the game. When an agent receives the `im_alive` message from another agent, e.g. `A` (indicated by the `source(A)` annotation), it adds the belief `alive(A)` to its BB, to say the agent receiving the message knows that `A` is alive. The bomber’s initial goal is `!initialize`, which involves waiting to receive `im_alive` messages from the other three agents in the environment. Once all messages are received, the game begins.

At this point agents execute a series of goals aimed at evaluating their current situation and making decisions accordingly. This means that they perceive their position, the presence of enemies, bombs, power-ups, wooden walls, and other relevant objects within a visual field. This information guides the agent’s strategy and actions at each step.

The agent’s behavior is inspired by the Simple Agent from the original Pommerman implementation [3]. At each state, the agent uses Dijkstra’s algorithm to compute distances to various objects within the game. To decide which action is best to take, the agent follows this logic:

1. It first checks if there is a power-up at its current position.

2. If there are imminent bomb explosions, the agent attempts to escape.
3. If the agent is adjacent to an enemy, it lays a bomb.
4. If an enemy is within three steps or a power-up is within two steps, the agent moves toward it.
5. If the agent is next to a wooden wall, it places a bomb to try to destroy the wall and create more open space.
6. If the agent is within three steps from a wooden wall, it moves towards it.
7. If none of the above conditions are met, the agent moves randomly to a new position.

Each time the agent moves, it recalculates the distances to objects and checks if the new position contains a power-up and if it is safe. We point out that the agent checks for power-ups at the current position using an *achievement goal*, as relying solely on the agent's percepts did not work. Indeed, every time the agent's percepts are updated, as when a cell (X,Y) contains a powerup (e.g. an *increase blast* power up), a corresponding percept `cell(X,Y,inc_blast)` is added to the belief base, which generate a new event. We tried using a plan with head `+cell(X,Y,inc_blast)` to handle this event, but looking at the debugger we noticed that the event was almost never selected by the agent. Then we added the achievement goal such that every time the agents moves to a new cell, it checks whether it has a power-up on it. A posteriori, we could observe that handling the powerups using percepts might have required a more complex customization of the `selectEvent()` function in the agent's class.

3.3 Interaction

As previously mentioned, at the start of the game, each agent broadcasts a message, `im_alive`, to inform other agents that it is alive. This broadcast also occurs when an agent dies from a bomb explosion, with the message `im_dead`. Upon receiving a message indicating that the agent A has died, each bomber agent first removes the belief `alive(A)` from its belief base, and then checks if there are any remaining enemies. If no enemies are alive, the agent declares itself the winner; otherwise, the game continues.

A crucial interaction occurs between the bomber agents and the detonator agents. When a bomber agent places a bomb, it sends a message to its corresponding detonator agent, i.e. `placed_bomb(X,Y,S)`, informing it of having placed a bomb with blast strength S at position (X,Y). Once the bomber agent that has placed a bomb moves to a new position and that position is safe, it instructs the detonator agent to achieve the goal of detonating the bomb. The detonator then proceeds to detonate the bomb and communicates back to the bomber agent that the detonation has occurred. The bomber can then update the `ammo` belief.

4 Implementation Details

As previously mentioned, our project extensively utilizes custom internal actions, which Jason supports by allowing extensions of the `DefaultInternalAction` class. In this section, we will briefly explain the functions of these actions.

`compute_distances.java`

This internal action is invoked whenever an agent changes its position on the map. As the name suggests, it computes distances from the agent's current position to other positions within a 5x5 grid representing the agent's visual field, which can be adjusted by modifying the `depth` parameter of the class. The computation uses Dijkstra's algorithm with the Manhattan distance function, as agents can only move up, down, left, or right. This internal action returns three different objects:

- **Items:** A dictionary that lists the positions of items within the agent's visual field.
- **Dist:** A dictionary that contains the distances between the agent and each cell within its visual field.
- **Prev:** A dictionary that holds the previous positions for each location within the visual field, enabling the construction of paths between locations.

In case of the presence of a bomb outside the visual field with a high blast range (having the explosion entering the visual field of the agent), the agent is not able to detect the safety of the cells where the explosion will take place. This is due to the fact that the visual field is 5x5. To overcome this scenario, where the agent can end up in a situation where bomb is too far to be detected, the first solution that comes to mind is to increase the depth to 10. However, computing the distance with respect to every object in a 10x10 visual field for each bomber every time they move can be quite heavy from a computation point of view, so we decided to stick with the visual field of dimension 5x5.

`is_position_unsafe.java`

This action is called whenever the agent moves to a new position or drops a bomb. Given the list of bombs in the visual field, it evaluates if the current agent position is unsafe. It returns a dictionary containing as keys the unsafe directions for the agent and as values the maximum bomb blast strength in that direction. Before being returned to the agent, it is converted into a Jason `ListTermImpl`, resulting in a list of lists. For example, consider an agent that has just dropped a bomb with a blast strength of 2. When `is_position_unsafe` is called, it will return the following Jason List:

```
[[Up,2],[Down,2],[Left,2],[Right,2]]
```

This means all directions in which the agent could move are potentially dangerous because the agent is exactly at the position of a bomb. Since the bomb has a blast

strength of 2, the maximum blast strength in each direction is also 2. For instance, if the agent decides to move right, it should move at least three steps to escape the bomb's explosion. As another example, if `is_position_unsafe` returns:

```
[[Left,2]]
```

This means there is a bomb with a blast strength of 2 to the left of the agent, so the agent should avoid moving in that direction.

find_safe_directions.java

Given the list of unsafe directions returned by `is_position_unsafe`, this action evaluates the safest direction for the agent to move to escape potential bomb explosions. The method `findSafeDirections()` works differently depending on whether all directions are unsafe or just some of them:

- **All Directions Unsafe:** If all four directions are unsafe, it checks if any direction leads to a position outside the bomb range using `isStuckDirection()`, which will return false if, by moving in a given direction, the agent will not find itself stuck. `isStuckDirection()` checks if moving in any unsafe direction allows reaching a position with a distance greater than the bomb blast strength (`dist > bombRange`). Alternatively, it could be a position outside the bomb scope (`nextX != positionX && nextY != positionY`). If the given direction satisfies one of these two conditions, `isStuckDirection()` will return false, and the direction will be returned as a safe direction in which the agent can move. If no such directions are found, `findSafeDirections()` will return `[SKIP]`, indicating that no movement is safe.
- **Some Directions Safe:** It makes three checks for each direction - if the resulting position is on the board, if it is a free cell, and if it is present or not in the unsafe directions dictionary. If a direction results in an out-of-board position, the position is added to the `disallowed` list. Safe directions are collected and returned. If no initially safe directions are found, it considers directions not in the disallowed list. If still no suitable directions are found, it returns `[SKIP]`.

Finally, the list of safe directions is converted to a format suitable for Jason and unified with the provided result term.

Other internal actions

- **is_object_in_radius.java** Checks whether a given object is within a specified range of the agent using Manhattan distance. If an object is within the specified range, it returns its location.
- **get_direction_to_object.java** Computes the direction to reach the location of a given object (returned by `is_object_in_radius.java`), based on the `Prev` dictionary.

- **is_safe_to_drop.java** Verifies whether it is safe to drop a bomb in the current position.

5 Self-assessment / Validation

We can say the project successfully implements the logic of a simple rule based agent playing Bomberman, meaning that the agents correctly move on the grid, detect nearby objects, place bombs when they are near to an enemy and try to escape when they drop a bomb or when one is detected nearby. However, it is often the case that agents dies because of their own bombs. This is likely because of the concept of visual field of the agents. Due to the visual field, agents can only see up to 2 cells from their current position, so if there is a bomb with a blast strength of 4 placed 3 cells away from the agent, it will likely be deadly.

6 Deployment Instructions

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

- Java version 19.0.2
- Jason CLI version 3.2.0
- Gradle version 8.0.1

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-morselli-vorabbi-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-morselli-vorabbi-ay2223
```

Now, execute the following command to run the Multi Agent System:

```
jason pommerman.mas2j
```

7 Conclusions

To recap, in this project we implemented an Agent Speak version of Bomberman, by using the Jason extension. This has been a tough challenge, especially considering the new programming paradigm to learn and the difficulty in implementing some of the mechanics of the game, such as the bomb explosions and the decision process of the bomber agent.

7.1 Future Works

Possible future works on this project could include the implementation of a team variant, in which the four bomber agents are split into two teams, and they have to eliminate the other team to win. Another possible extension could be the integration of a Reinforcement Learning framework to make the agent learn from their experience. However, as we stated when working at this project, implementing RL directly in our code could be quite difficult. The project could then be used as a way to simulate games between agents and to gather a dataset of examples composed of the state of the environment and the action taken by the agent in such state, which can be then used to train a NN. Deep Q-Learning from Demonstrations (DQfD) [2] [4] is a recent method that leverages small sets of demonstration data to accelerate the learning process, and could be a possible option. However, even the integration of such approaches with our projects still seems non trivial and has to be evaluated more carefully.

7.2 What did we learned

Throughout this project, we not only learned a new programming language, AgentSpeak, but also encountered new challenges and gained a fresh perspective on programming through the agent-oriented paradigm. This was particularly significant as it was our first experience working with Multi-Agent Systems, a topic we had not explored before.

One of the major challenges we faced during development was the limited and often insufficient documentation available for Jason. Indeed, the API documentation for many Jason classes unfortunately lacked detailed comments, which frequently left us guessing about the functionality of certain methods. This lack of clarity slowed down the development process.

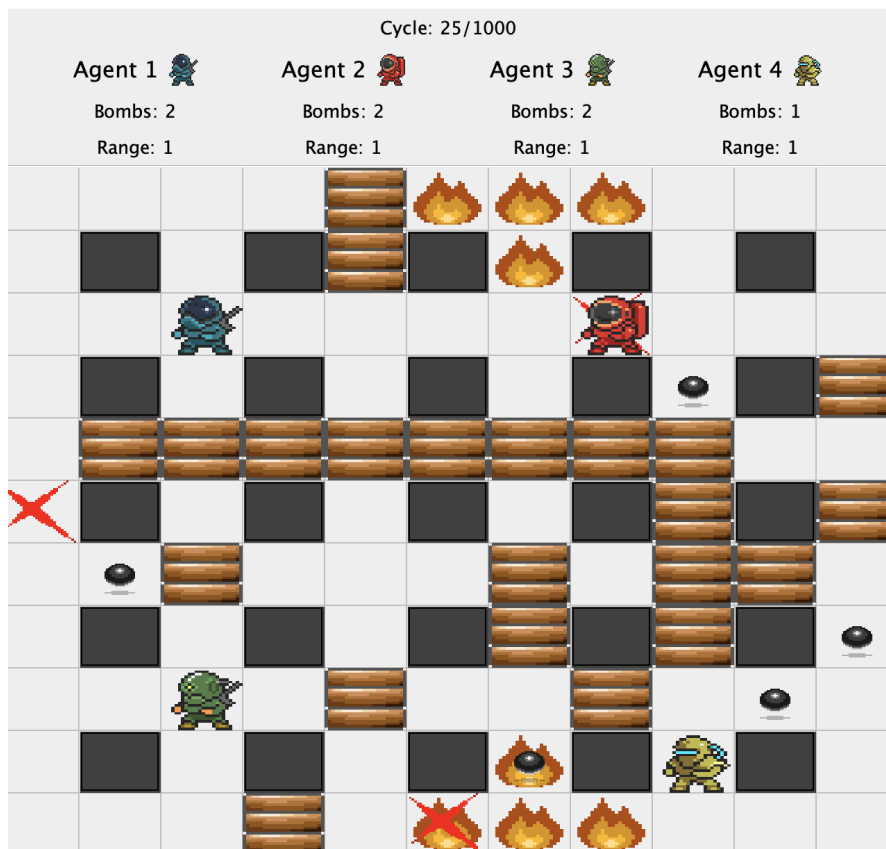
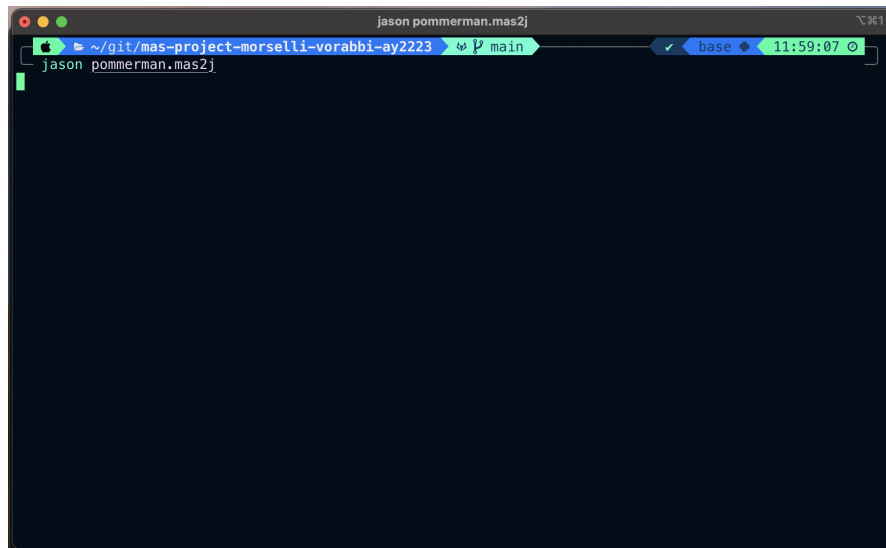
Additionally, during development, we discovered a bug in AgentSpeak's behavior, which we promptly reported to our professors. This experience highlighted the importance of robust documentation and community support when working with niche programming frameworks.

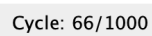
Despite these obstacles, the project provided valuable insights into both agent-based programming and the complexities of Multi-Agent Systems. In the end, although we did not achieve the goals we had initially set ourselves, due to a higher technical complexity than expected, we are quite satisfied with our work.

References

- [1] R.H. Bordini, J.F. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak using Jason*. Wiley Series in Agent Technology. Wiley, 2007.
- [2] Chinmay Talegaonkar Dhruv Shah, Nihal Singh. Multi-agent strategies for pommerman. <https://people.eecs.berkeley.edu/~shah/docs/pommerman.pdf>.
- [3] Cinjon Resnick et al. Playground. <https://github.com/MultiAgentLearning/playground>.
- [4] Todd Hester, Matej Vecerík, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Andrew Sendonaris, Gabriel Dulac-Arnold, Ian Osband, John P. Agapiou, Joel Z. Leibo, and Audrunas Gruslys. Learning from demonstrations for real world reinforcement learning. *CoRR*, abs/1704.03732, 2017.
- [5] Rafael H. Bordini Jomi F. Hübner. Jason interpreter 3.3.0-snapshot api. <https://jason-lang.github.io/api/>.
- [6] Cinjon Resnick, Wes Eldridge, David Ha, Denny Britz, Jakob Foerster, Julian Togelius, Kyunghyun Cho, and Joan Bruna. Pommerman: A multi-agent playground, 2022.
- [7] Wikipedia. Bomberman. <https://en.wikipedia.org/wiki/Bomberman>.

Appendix A - Usage Examples



Agent 1 

Bombs: 4

Range: 3

Agent 2 

Bombs: 4

Range: 3

Agent 3 

Bombs: 3

Range: 1

Agent 4 

Bombs: 5

Range: 1