Autonomous Software Agents Project Report

Bonetto Stefano
247179
stefano.bonetto@studenti.unitn.it

Roman Simone
247181
simone.roman@studenti.unitn.it

Abstract

This project develops an autonomous software agent designed to play a game, aiming to maximize points by collecting and delivering parcels. Using a Belief-Desire-Intention (BDI) architecture, the agent senses the environment, manages beliefs, activates goals, selects plans, and executes actions to achieve its objectives. Initially focusing on a single agent's capabilities, including automated planning and belief revision, the project later extends to include a second agent, enabling communication and cooperative strategies.

1 Introduction

In this report, we present the key components of our autonomous agent, designed to play the Deliveroo game. The objective of the game is to earn points by collecting and delivering parcels to designated "delivery tiles". To achieve this, we've implemented a Belief-Desire-Intention (BDI) architecture, which allows the agent to perceive the environment, establish goals, select the best plans from a set of predefined options and execute actions. This architecture enables the agent to continuously update and refine its plans to successfully accomplish its objectives.
The project is divided into two main parts:

- **Single Agent**: the first phase focuses on building a single autonomous agent capable of understanding and processing information from its environment. The agent updates its beliefs, sets goals, forms intentions and takes actions accordingly.
- **Multi Agent**: in the second phase, we extend the project to include a second autonomous agent. This extension enables the agents to share information about their respective beliefset and exchange goals. We have implemented a *MASTER-SLAVE* model, where the *MASTER* agent is responsible for selecting the best actions not only for itself but also for the *SLAVE* agent.

We evaluate and validate performances of agents by conducting a series of simulations across different levels, each presenting unique challenges and characteristics.
In this report, we will discuss the logic, strategies, and structure behind the development of both the single-agent and multi-agent systems.

2 Structures

In our code, we simplify the management of information related to the environment and agent sensing by using two main structures: `AgentData` and `Map`. These classes encapsulate all relevant information, making the code more manageable.

2.1 AgentData

We use this class to store the agent's data, which is crucial for encapsulating all of the agent's beliefs and options. This class becomes especially important in multi-agent scenarios in fact agents can easily exchange information by sharing their instance of this class, which simplifies coordination and interaction between them. In such cases, the `MASTER` agent maintains two instances of this class: one for itself and one for the `SLAVE` agent. This setup allows the `MASTER` to have a clear understanding of both agents' intentions and perceptions, enabling it to make more informed decisions.
`AgentData` is composed of the following elements:

- `name`: name of the agent
- `id`: id of the agent
- `pos`: position of the agent, updated with the `onYou` asynchronous function.
- `role`: role of the agent, this could be:
  - `NOTHING` in single-agent case.
  - `SLAVE` or `MASTER` in multi-agent case
- `parcels`: list of parcels sensed using `onParcelSensing` asynchronous function (see Section 3.3)
- `parcelsInMind`: list of parcels stored in the agent's head (i.e. those that have already been picked up but not yet put down). By keeping track of these parcels, the agent can evaluate its current standing and plan its next moves more effectively.
- `options`: store the list of option that are formulated (see Section 4.1).
- `best_option`: represents the best option from the previous list with respect to utility. In a multi-agent scenario, the `MASTER` calculates the optimal options for both itself and the `SLAVE` (see Section 4.1.1).
- `adversaryAgents`: this is the list of agents updated based on the agent's perception (see Section 3.4). In a multi-agent scenario, it also includes enemies detected by other agent.

2.2 Map

This class is essential for managing and updating map information.
The most important attributes are:

- `original_map`: the initial map provided by the server remains unchanged throughout the program and serves as a reference for restoring the map's original state when needed.
- `map`: the map used by the agent to plan its behavior. This version of the map includes additional data, such as information about sensed agents (see Section 3.4). The class provides methods to reset the map to its original state (`resetMap()`) and to update specific values on the map (`updateMap(x, y, value)`).
- `width` and `height`: dimensions of the map, used in various calculations.
- `parcel_reward_avg`, `parcel_observation_distance`, and `decade_frequency`: configuration parameters that depend on the level being played. These include the average reward of parcels, the maximum distance within which the agent can sense a parcel, and the frequency of decades (see Section 3.1).
- `deliveryCoordinates`: the coordinates of each delivery tile. This array is populated with all delivery coordinates at the beginning of the gameplay and is never modified during it.
- `spawningCoordinates`: the coordinates of each spawning tile guide the agent's movements during random actions. For each spawning coordinate, an additional metric called the *score* is calculated. This score is determined by creating a window centered on the spawning coordinate, with a size equal to `parcel_observation_distance`. The score is calculated by counting the number of spawning tiles within this window. A higher score indicates a more favorable spawning tile.
- `myBeliefset`: It stores information about the existence and relationships between tiles (e.g. from a given tile, which adjacent tiles are immediately reachable). It's updated based on the current state of the map (via the `updateBeliefset()` method). This information is used for formulating the PDDL problem (see Section 5).

3 Belief

An agent's understanding of its environment depends on its beliefs, which are shaped by its perceptions. These beliefs are not static, they are asynchronously refined and updated as the agent encounters new stimuli. This dynamic process ensures that the agent's view of the world remains up-to-date and adaptable.
The key perceptions that our agent is capable of detecting include:

- Configuration parameters of the current level
- Map
- Parcels' sensing
- Agents' sensing

3.1 Configuration parameters

Whenever a new game begins, you can select a level from those available on the server. Each level comes with specific fixed parameters that we use to optimize our agent's performances. These parameters are extracted using the asynchronous function `onConfig`.
First of all we want to compute the so called `decade_frequency` which will be useful when computing utility for the parcels:

$$decade\_frequency = \frac{agent\_movement\_duration}{parcel\_decading\_interval} \tag{1}$$

This ratio quantifies the rate at which a parcel's score diminishes with each step the agent takes, helping to align the measurement units between the agent and the parcel.
In certain levels, the `parcel_decaying_interval` is set to `"infinite"`; in such cases, we assign it a value of `Number.MAX_VALUE`.
Lastly, we extract the `parcel_reward_avg` that we'll use to determine a threshold for the `go_put_down` intention (see Section 4.1.3).

3.2 Map intialization

At the beginning of each game, the agent receives a JSON file from the server, which includes information about each non-empty tile on the map. This file is used to construct a matrix that represents the map more efficiently. Each tile in the matrix is assigned one of the following values:

0	Empty tiles
1	Walkable non-spawning tiles
2	Delivery tiles
3	Walkable spawning tiles

This matrix data is then used to populate the fields in the `Map` class, as described earlier (see Section 2.2).

3.3 Parcels' sensing

Parcel detection is handled by the asynchronous function `onParcelsSensing`, which is triggered whenever a parcel is detected. For each detected parcel, we records its location (`x` and `y` coordinates), reward value and the timestamp of the most recent sighting.
Next, we examines previously recorded parcels that are no longer present in the new perception set. For each of these parcels, we calculate the elapsed time since the last update and we reduce the value of the reward based on this time interval. If the reward remains above a certain threshold and the elapsed time is less than 15 seconds, the parcel is updated (the reward and timestamp) and added to the list.
Finally, the parcels' list in `agentData` is refreshed to include the newly updated parcels.

3.4 Agents' sensing

The asynchronous function `onAgentsSensing` is responsible for updating the information about opposing agents detected by the agent. Initially, the map is reset to its original values.
For each perceived agent, the function records a timestamp to indicate when the agent was last seen. If the agent is not already in the `adversaryAgents` list within the `agentData` class, it is added to the list with its direction initially set to *'none'*. If the agent is already in the list, the function calculates its direction of movement by comparing its current position with its previous position. If movement is detected, the direction is updated accordingly (*'right'*, *'left'*, *'up'*, or *'down'*).
Once the direction is updated, the agent's information in the list is replaced with the new data.
Finally, the `map` is updated to reflect the current positions of all opposing agents, marking their positions and directions as walls (we set these tiles' values as -1). This ensures that the agent can navigate the map avoiding collisions with adversaries.

3.4.1 Agent Collisions

Unfortunately, this isn't sufficient to completely avoid collisions between agents, as they are likely moving. The main issue with our implementation is that once the PDDL plan is computed and the corresponding path is returned, the agent starts moving (as if it's a horse with blinders on) and we're unable to check for potential collisions with other agents at each step. We've tried various techniques to prevent and manage collisions between both cooperative and adversarial agents.
First, to avoid collision between cooperative agents, we decided to try a very invasive technique: for each path computed for the `SLAVE`, we reset the map and then set the entire path as unused tiles (assigned a value of -2) on the `MASTER` side, effectively removing these tiles from the `MASTER`'s path calculation.

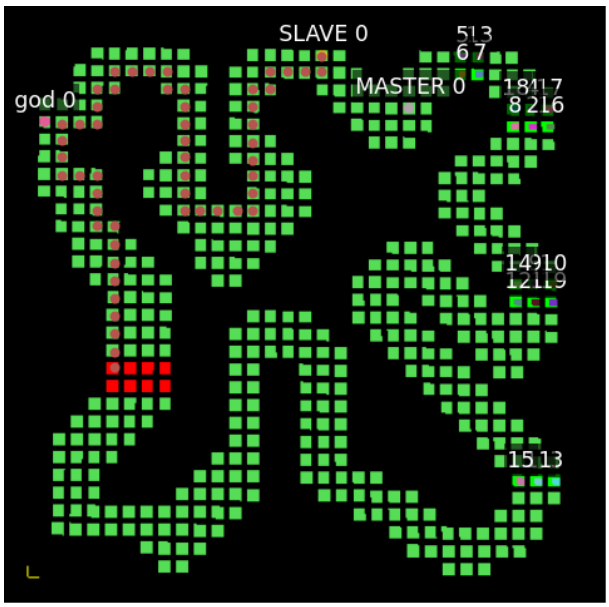

Figure 1: *The `SLAVE` intends to put down parcels and begins following the red path. If we supposed that the `MASTER` wants to put down his parcels, it is forced to take a longer route around the map. This results in an highly suboptimal solution, significantly reducing the overall score.*

As we've experienced (e.g. see the figure above), if the `SLAVE` completely blocks the `MASTER`'s path, or, as shown in the figure, blocks the optimal path to the `MASTER`'s goal, it can lead to inefficient or even unfeasible solutions.
So we opt for a different technique, we simply managed the collision once it happens: when the two agents collide, they attend for a few seconds: the `MASTER` waits 7 seconds and the `MASTER` waits 2 seconds.
While this approach does slightly slow down the overall process and movement of the agent, it is the most efficient solution among those we have tried.

4 Option and Intention Loop

The architecture in which the agent operates follows the classic **Belief-Desire-Intention (BDI)** model. In this framework, the agent generates several potential options based on its beliefs and desires, then filters these options by evaluating their utility function values to determine the most suitable action.

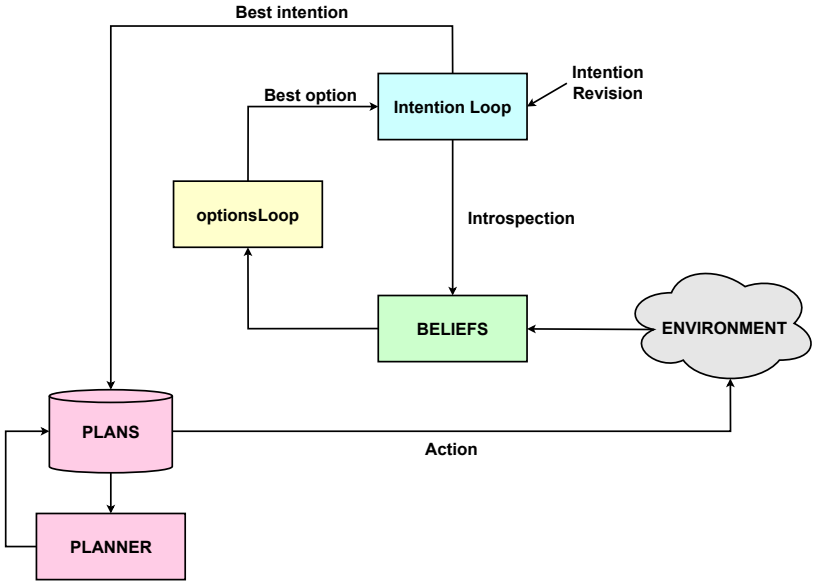

Figure 2: *BDI loop*

As illustrated in Figure 2, our approach utilizes two distinct loops: the Option Loop and the Intention Loop.

4.1 optionsLoop

The `optionsLoop` is triggered every second. It is designed to generate and evaluate a series of options for the agent and choose the best one, operating within a continuous decision-making loop. This process is crucial for determining the best possible action based on current circumstances and available information. Three types of options are evaluated:

- **pick-up**: the agent examines the parcels stored in the `parcels` list of the `agentData` class. Each parcel is filtered based on specific criteria: it must have a reward value greater than 3, it must not have been picked up by another agent, and it must be located on an accessible tile of the map. If a parcel meets these criteria, the utility of picking it up is calculated using `calculate_pickup_utility` (see Section 4.1.2). If the utility is positive, the option of picking up the parcel is considered.
- **put-down**: the agent recursively evaluates whether to deliver the collected parcels based on an ad hoc utility function that determines if delivering them is beneficial. If the parcel decay interval (`parcel_decading_interval`) is not infinite, its utility is computed using the `calculate_putdown_utility` function (see Section 4.1.3), which also determines the best delivery location. The resulting option is added to the `options` list. If the parcel decay interval is infinite, the agent decides to deliver the parcels whenever the '*inmind*' score (`get_inmind_score()`) exceeds a certain threshold, based on the average parcel reward multiplied by a factor (`MULTIPLIER_THRESH_GO_PUT_DOWN`).
- **go-random**: this option is always included with a predefined utility of 2 and is triggered when the agent has no parcels to pick up or deliver. It involves moving towards the spawn area with the highest score (see Section 2.2), or if the agent is already in that area, heading to the furthest delivery point (to enhance exploration over exploitation). In the multi-agent scenario, the `SLAVE` behaves as previously described, while the `MASTER` moves toward the furthest delivery tile w.r.t. the `SLAVE`. This strategy is intended to maximize the distance between the two agents, promoting more efficient coverage of the map.

4.1.1 best_option choice strategies

Once the list of options is formed, the next step is to choose the best one. This process differs significantly between the single-agent and multi-agent scenarios.
In the **single-agent** case, the agent simply selects the option with the highest utility.
However, in the **multi-agent** case, the selection process requires more careful considerations. The primary goal in this case is to ensure that both agents can operate effectively without interfering with each other's tasks.
Here's how the choose of best option works (only Master do these operations):

1. **Unification of options**: the process begins by unifying the available options for both agents. The `SLAVE`'s options are added to the `MASTER`'s list, and vice versa. This step is essential because it allows each agent to be aware of the other's possible actions. If an option to pick up a parcel exists for only one agent, we evaluates whether the other agent could also pursue that option. If it's possible, the parcel's reward is intentionally reduced through a penalty, making it less attractive (this penalty encourages the agents to maintain a distance from each other by targeting different parcels), finally we compute the utility for that parcel. After that, for each option we obtain two utility values, each corresponding to one of the two different agents.

2. **Best option selection**: after combining the options, we determine the `best_option` for each agent by evaluating the utility of each available choice for both the `MASTER` and `SLAVE` based on their respective utilities.

3. **Conflict resolution**: once the best option are identified, we need to checks for potential conflicts between the agents. If either agent's `best_option` involves moving to a **go-random**, no further action is needed since this option inherently avoids conflicts. However, if both agents decide to **put-down** parcels at the same location, there might be collisions. `MASTER` compares the utilities of both agents for that option and redirects the agent with the lower utility to the nearest alternative delivery point. This adjustment ensures that both agents can deliver their parcels without overlap. Similarly, if both agents choose to **pick-up** the same parcel, we resolve this conflict by allowing the agent with the higher utility to keep that option while redirecting the other agent to pick up a different parcel.

4.1.2 Pick-up Utility

The *pick-up utility* is designed to evaluate the effectiveness of picking up a specific parcel by considering several factors, such as the distance to the parcel, the distance to the delivery point, the number of parcels in mind, and the presence of opposing agents. This utility assesses the potential final reward by factoring in the time required to pick up the parcel and complete the delivery, while also incorporating a penalty for the presence of closer opponents.
The pick-up utility is computed using the following formula:

$$utility = RewardParcel + RewardInMind - [f \times dist(me, delivery) \times (nParcelsInMind + 1)] \quad (2)$$

where:

- f is the decade frequency (see Section 3.1).
- $RewardParcel$ is the reward of the parcel upon reaching its location, defined as:

$$RewardParcel = scoreParcel - f \times dist(me, parcel) \quad (3)$$

- $RewardInMind$ is the reward of the parcels currently in mind when reaching the parcel location, defined as:

$$RewardInMind = scoreInMind - [f \times dist(me, parcel) \times nParcelsInMind] \quad (4)$$

- $dist(me, delivery)$ is the distance from the agent to the delivery point.
- $nParcelsInMind + 1$ is the number of parcels currently in the agent's possession plus the parcel we are considering to pick up.

Additionally, a penalty is applied if the distance between the agent and the parcel is bigger than the distance between the nearest opposing agent and the same parcel. This penalty is incorporated as follows:

$$utility = utility - [malus \times (dist(me, parcel) - dist(nearestAgent, parcel))] \tag{5}$$

where the *malus* term represents a constant penalty factor. In our implementation, the value of *malus* is set to 0.7, which provides an effective balance in the utility calculation.
All distance calculations, including $dist(me, parcel)$, $dist(me, delivery)$, and $dist(nearestAgent, parcel)$, are performed using Breadth-First Search (BFS) due to its faster pathfinding capabilities compared to PDDL.

4.1.3 Put-down utility

The *put-down utility* is designed to evaluate the effectiveness of delivering parcels to the nearest delivery point from the agent's current position. This utility quantify if it's better to deliver parcels now or wait for a chance to get a better deal later.
The utility is computed using the following formula:

$$utility = scoreInMind - (f \times dist(me, nearestDelivery) \times nParcelsInMind) \tag{6}$$

where:

- f is the decade frequency (see Section 3.1).
- $scoreInMind$ represents the total reward the agent has accumulated from the parcels it is currently carrying.
- $dist(me, nearestDelivery)$ denotes the distance from the agent's current position to the nearest delivery point. This distance is calculated using Breadth-First Search (BFS), for the same motivations as the pickup utility.
- $nParcelsInMind$ is the number of parcels currently carried by the agent.

To address the issue of low utility values resulting from this calculation alone, a multiplier of 1.5 is introduced to encourage deliveries:

$$adjustedUtility = 1.5 \times utility \tag{7}$$

4.2 Intention loop and intention revision

The intentionLoop is responsible for continuously processing the agent's intentions. At each iteration, it checks if there are any intentions in the `intention_queue`. If the queue is not empty, it pops current intention from the queue and attempts to achieve it. If the intention fails to be achieved, an error is caught, and the failed intention is removed from the queue to ensure the agent does not repeatedly attempt it.
When an intention is successfully achieved, additional steps are taken based on the nature of the intention to ensure that the agent's internal state accurately reflects the outcomes of its actions:

- If the intention is to **pick up** a parcel, the parcel ID is added to *parcelsInMind*, keeping track of parcels that the agent has targeted.
- If the intention involves **put down** parcels, all parcels listed in *parcelsInMind* are removed from *parcels*, and the list is reset.

Every time a best option is choosen in the optionLoop we do the intention revision before pushing this option in the `intention_queue`. First we check whether the new intention already exists in the queue by comparing it with the existing intentions. If an intention with the same details is found, its utility value is updated. If the intention is not already in the queue, a new `Intention` object is created and added to the queue.
Subsequently, the `intention_queue` is sorted based on the utility values of the intentions. The sorting process uses the **bubbleSort** method, which organizes the queue so that higher-priority intentions are processed first.

After updating the queue, we have the **intention revision** step. We check if the newly added or updated intention is of higher priority than the current one. If a switch is needed, we stop the current intention and do the transition. This allows the agent to adapt dynamically to changes in priority and ensures that it is always working towards the most beneficial goal.

5 Planning

One of the most crucial aspects of our agent's functionality is the action planning process, which involves determining the optimal path to achieve a given goal.
Initially, we implemented a basic planning strategy using Breadth-First Search (BFS) to explore and identify the most efficient path from the agent's current position to the target goal.
Following the initial implementation with BFS, we enhanced our planning capabilities by integrating Planning Domain Definition Language (PDDL).

5.1 Initial Version - BFS

Breadth-First Search (BFS) is a fundamental algorithm to explore graphs or grids. It starts from the initial node and systematically explores all neighboring nodes at the current depth level before progressing to nodes at the next depth level. This approach guarantees finding the shortest path in unweighted grids where each movement step has an equal cost.
Initially, we used this algorithm to calculate the pickup utility, putdown utility, and to find the nearest delivery point. Additionally, BFS was employed to determine the path for the agent's movements. Subsequently, we change our implementation by using Planning Domain Definition Language (PDDL) for pathfinding tasks, but we continue to use BFS for the other tasks due to its superior speed in these scenarios.

5.2 PDDL

A **PDDL-based planner** generates a plan by interpreting a given problem description, which consists of a sequence of actions that transition the system from the initial state to the desired goal state. The planner explores potential actions while considering their preconditions and effects, constructing a feasible plan to achieve the specified objectives.
In our implementation, we have defined a fixed domain for our PDDL problems. This domain specifies the allowable predicates, their implementations and their associated parameters (preconditions and effects). As mentioned earlier, PDDL is used primarily for calculating the path during the movement phase. Before this, we update the PDDL problem using the most recent map, which includes the positions of other observed agents.
We use a function that updates the agent's beliefset by iterating through the map and declaring possible movements based on the presence of neighboring walkable tiles. It ensures that the PDDL problem accurately reflects the agent's environment including the latest observations.

6 Communication

A fundamental aspect of the multi-agent scenario is the communication between the agents.
The main idea is to have a *MASTER-SLAVE* architecture where the `SLAVE` shares his beliefset with the `MASTER` which is responsible of incorporating options, enemies and parcels from both agents and choosing the `best_option` for both of them (see 4.1.1).
Our communication pipeline relies on two types of messages: handshake messages and belief/option-sharing messages.

6.1 Handshake

Initially, to establish communication in our multi-agent system, we implemented an handshake process to synchronize and identify the roles of the agents.
The first agent that established connection with the server (*agent1*) sends a broadcast message containing its identity and a tag indicating the start of the handshake. In our implementation, the broadcast message is:

```
'[HANDSHAKE] ' + client.name + ' firstMessage'
```

The second agent (*agent2*) received the first agent message and verifies its format. If the message matches the expected pattern, *agent2* establishes the connection.
At this point, *agent2* assumes the role of `SLAVE` and it sends an acknowledgment back to the first agent (in our case the message is simply `'[HANDSHAKE] ' + client.name + ' ack'`).
Upon receiving the acknowledgment, *agent1* identifies itself as the `MASTER`. This exchange completes the handshake, ensuring both agents are aware of their roles and they know their respective IDs to send future messages.

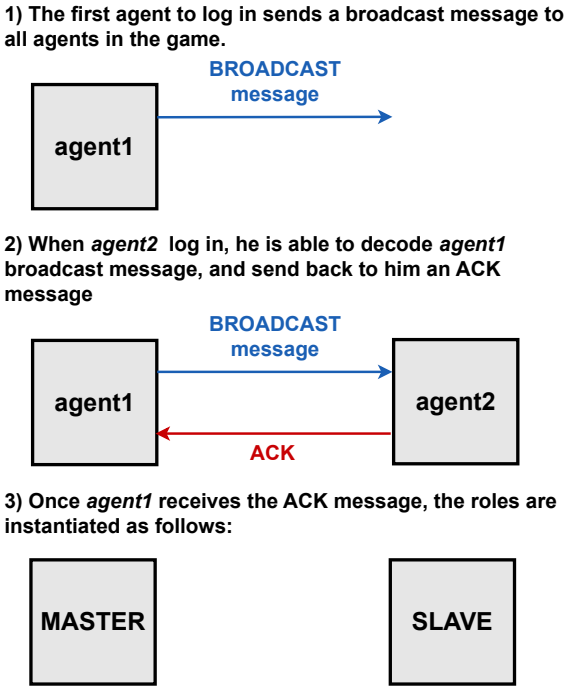

Figure 3: *Handshake procedure*

6.2 Belief/options sharing

As previously mentioned, the process of determining the best options for both the `SLAVE` and the `MASTER` agents is centralized and conducted on the `MASTER` side.
To implement this type of collaboration, an exchange of information between the two agents is continuously repeated. In particular:

- `SLAVE` sends his `AgentData` structure that includes relevant data such as its position, options, and other information that could influence decision-making. When it receive back an update from the `MASTER`, it updates his `AgentData` structure.
- `MASTER` instead, receives new data by the `SLAVE` and stores it into `CollaboratorData`. Then it updates his `adversaryAgents` list with the ones observed by the `SLAVE` (no need of handling parcels because we managed them directly from the option's list).

  After computing strategies for both the agents (see Section 4.1.1), it sends back to the `SLAVE` his beliefset updated with his knowledge and his `best_option`.

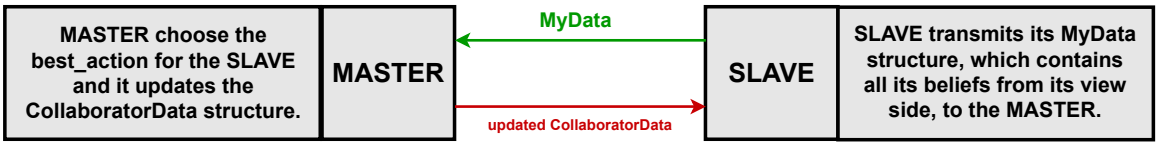

Figure 4: `MASTER-SLAVE` interaction

7 Benchmark and results

As we've mentioned before, we've run some tests on different levels of the Deliveroo.js server.
The server proposes two sets of levels, each with 9 stages: one for single-agent gameplay and one for multi-agent gameplay.
All the tests we conducted were based on 5-minute gameplay sessions. For each test, we utilized a PDDL planner, which we executed using the planning-as-a-service repository.
This repository simplifies the process by allowing us to run the PDDL planner locally within a Docker container. It also gives us the flexibility to choose the number of workers, making it easier to scale and manage the planning tasks efficiently.

Level	Score
24c1_1	550
24c1_2	927
24c1_3	1253
24c1_4	743
24c1_5	2399
24c1_6	399
24c1_7	165
24c1_8	170
24c1_9	965

Table 1: *Results in single-agent scenario*

Level	Total score	A1 score	A2 score
24c2_1	670	250	420
24c2_2	1259	742	517
24c2_3	1231	706	525
24c2_4	2854	1294	1559
24c2_5	1726	848	878
24c2_6	1998	955	1043
24c2_7	1512	675	837
24c2_8[1]	381	0	381
24c2_9	1699	838	861

Table 2: *Results in multi-agent scenario*

8 Conclusions and future improvements

In this project, we have successfully developed an autonomous agent system that can navigate the challenges of parcel collection and delivery within a dynamic environment. Using the Belief-Desire-Intention (BDI) architecture, our agents demonstrated strong capabilities in making decisions, adjusting their strategies, and effectively coordinating with each other.
The implementation of the `MASTER-SLAVE` architecture combine simplicity and effectiveness in managing coherently the behaviours of the agents. This cooperative approach proved particularly beneficial in managing complex scenarios where `MASTER` is able to guide the `SLAVE` into the best action to maximize the overall score.
Overall, we are pleased with the results.
However, we faced challenges with the system's responsiveness, due to the PDDL solutions aren't generated in real-time. This delay can affect the agent's performance in fast-paced environments where quick decisions are crucial. To improve real-time capabilities, one idea is to send only a specific region of the map to the PDDL solver, though this might lead to less optimal solutions.
In summary, while our project has achieved its primary goals, there is room for improvement. We look forward to exploring these areas in future work to build upon the solid foundation we have established.

[1] Due to the complexity of the process and the low average reward of the parcels, we decided not to use PDDL as it would be too time-consuming. Instead, we employ our original approach using BFS, which is more efficient for this scenario.