Autonomous Software Agents Report

Marina Segala
248447
marina.segala@studenti.unitn.it

Pietro Bologna
248715
pietro.bologna@studenti.unitn.it

Abstract

The goal of the project was to develop an autonomous software agent designed to play the Deliveroo.js game, maximizing points by collecting and delivering parcels. Using a Belief-Desire-Intention (BDI) architecture, agents sense the environment, manage beliefs, activate goals, select plans and execute actions to achieve objectives. The first part is done based on a single agent's capabilities. Then the project was extended to working in a team with a second agent, enabling communication and cooperative strategies.

1 Introduction

This report contains the key components of our autonomous agents, designed to play the Deliveroo game. The goal of the game is to earn points by collecting and delivering parcels to designated delivery points. To achieve this, we've implemented a Belief-Desire-Intention (BDI) architecture, that allows the agent to perceive the environment, establish goals, select the best plans and execute actions. This architecture allows the agent to continuously revise and improve its plans to effectively achieve its objectives.

The project is structured into two main parts:

- **Single Agent:** a single autonomous agent is developed, capable of interpreting and processing information from its environment, updating its beliefs, forming its intention and taking actions.
- **Multi-Agent:** the project is expanded to include a second autonomous agent, enabling the agents to share information and coordinate their actions. In this case, the basic requirements include a game strategy and a coordination mechanism. Specifically, agents must exchange information about the environment and their mental states to collaborate effectively and develop plans.

Finally, we assess and validate the performance of the agents by running a series of simulations at various levels, each with its own set of challenges and characteristics. This report will explore the logic, strategies, and structure involved in the development of both the single-agent and multi-agent systems.

2 Structures

Two main structures, `Me` and `Map`, are used to gather information about the agent and its environment, ensuring that all relevant knowledge is organized efficiently. Using these two classes, the object `myAgent` is created with `IntentionRevions`, enabling effective decision-making and adaptation.

2.1 Me

The `Me` class represents the agent's state, encapsulating all essential information about its current status. Specifically, it consists of:

- `id`: id of the agent;
- `name`: name of the agent;
- `x` and `y`: coordinates of its position, updated with the `onYou` asynchronous function;
- `score`: the agent's current score;
- `particelsCarried`: boolean flag for saying if the agent is carrying something;
- `map_particles`: classic Map structure for saving the particles' position;
- `master`: boolean flag, for understanding the role of the agent. In particular, if it is `true`, the agent is the *MASTER*; otherwise, the agent can be a *SLAVE*, a *enemy* or we are in the *Single Agent Scenario*
- `friendId`: id of the other agent that collaborates with;
- `currentIntention`: the current intention the agent is executing.
- `notMoving`: boolean for understanding if the agent is executing some intention or is not moving
- `counterFailer`: counter for how many times in a row the agent fails to execute an intention
- `prevPos`: coordinates of the agent's previous position of the `pick_up` move;

2.2 Map

Another created key class is `Map`, which represents crucial information about the environment. This structure enables the agent to process and update its surroundings effectively. The components of the `Map` include:

- `width`: width of the map provided by the server
- `height`: height of the map provided by the server
- `map`: list of coordinates that can be reached by the agent. They can be delivery points or simple tiles;
- `deliverPoints`: list of coordinates of delivery points

- `value_coords`: list where each point on the map corresponds to 0 (not walkable) or 1 (walkable)
- `agent_beliefset`: used for PDDL problem, it contains information about the existence and relationships between tiles

Besides functions used for saving or printing values, in `Map` there are also:

- `getPossibleDirection(x, y)`: it returns the possible moves that can be done starting from a specific point (left, right, up, down)
- `update_beliefset()`: it updates the `agent_beliefset`, based on the current state of the map

3 Belief of the agent

The agent's knowledge is built from its interactions with the environment. Its belief is continuously refined and updated asynchronously whenever new stimuli are encountered, ensuring that the information remains up to date.
The main observations that our agent can detect include the following:

- Map
- Particle' sensing
- Agents' sensing

3.1 Detection of Map information

The information obtained from the map does not directly be part of the agent's belief: it is used to carry out the operations needed for performing the selected intention. The general information derived from the entire map is not taken into account in this decision-making process in determining the most appropriate action.
As described in the Sec 2.2, the saved parameters have a very important role during different steps, such as:

- the calculation of supported functions, like `findNearestDeliveryPoint` or `isValidPosition`
- update of the `BeliefSet`
- understand the possible moves agents can do in a deadlock
- help saving the position of other agents

3.2 Particle' sensing

The parcel sensing mechanism is handled asynchronously through the `onParcelsSensing`. When new parcels are detected, their data, including the ID, position (x and y coordinates are updated and stored. Each time a parcel is perceived, the agent's internal knowledge is refreshed, ensuring it has the most up-to-date information about its environment.
For each perceived parcel, the agent checks whether it is available for pickup (not already carried by another agent). If the parcel meets the criteria, it is added to the list of potential actions,

which are prioritized by the distance between the agent and the parcel. This ensures that the agent will try to pick up the closest parcels first.
Furthermore, when multiple suitable parcels are found, the agent can send information about the available parcels to its teammates, as described later (see Section 6.2).

3.3 Agents' sensing

The agent sensing is managed asynchronously through the `onAgentsSensing` event listener. This function updates the agent's knowledge about other agents in the environment by storing and maintaining their information in the `agents_map`. Each time an agent is detected, the time of detection is recorded, and the agent's position (x and y coordinates) is updated in the map. If an agent has not been seen in the last 20 seconds, it is removed from the map to ensure that only currently relevant agents are being tracked. This prevents outdated data from affecting the decision-making process.
Additionally, as better described in Section 6.2 if the agent has a teammate and other agents are detected, information about these agents is shared with the teammate, allowing for better coordination and collaboration.

4 Intention Loop

The agent functions within a framework grounded in the classic Belief-Desire-Intention (BDI) model. In this setup, the agent generates various potential actions driven by its beliefs and desires. A key focus of our implementation was determining the optimal plan and action to execute, ensuring effective decision-making.

4.1 Intention Loop

The *IntentionLoop* is responsible for continuously processing the intentions of the agent. At each iteration, if at least one intention is present in a queue, the last one is selected and a check is done to understand if it is still valid. In the case of a positive answer, it tries to achieve the chosen intention. In any case, after all these operations, the intention is removed from the queue and the time of the last move is saved as the current timestamp. Saving the time of the last move is important to understand how long our agent has been standing still: after 6 seconds, the agents will move randomly on the map.
The function `moveToRandomPos` is also used if there are no intentions to achieve. When the predicate of the intention is calculated with this method, no other actions are done in between: the agent has to finish the random move and then it can perform all the other intentions.

In general, the best option available is taken into account when creating the intentions queue, such as trying to find the nearest particle to pick up or the closest delivery point to deposit it. More specifically, these are the steps followed for the creation of the queue:

- Further check to make sure to choose the best intention to achieve
  - the action `pick_up` of nearest particle has to be found
  - if the agent is carrying some particles (`me.particelsCarried = True`) and the closest delivery point has a lower distance than the nearest particle, the action `put_down` is created

  - if the action of our intention is not defined, the action `go_to` is created (based on a random move)

- If the intention is not already present in the queue, it is added
- The queue is sorted according to the lower distance from the agent, as the last step

In the event that the agent is unable to complete the selected intention for some reasons, the internal counter for failed actions will increase. In this case, if the agent is carrying some particles, a `put_down` is executed immediately.
After 2 times in a row with no satisfied intentions, the agent will not move randomly anymore, but it will go to the `prev_pros` coordinates.

5 Planning

The strategy used to move agents is based on a structured planning approach, intended to optimize path-finding and decision-making processes. Strategies utilizing **Breadth-First Search (BFS)** and **Planning Domain Definition Language (PDDL)** have been developed.
The primary goal of this implementation is to capture as many particles as possible without getting stuck: not so much importance was given to the rewards and the possible cost of reaching a certain destination. The main concept behind it is to move randomly to increase the chances of increasing the agent's internal score in another location on the map if it cannot do anything more than an arbitrary period of time.
To enable or disable the use of PDDL, the user must set the `usePDDL` variable to `true` in the `intention.js` file.

5.1 BFS

According to the definition of BFS, the chosen strategy is to find the shortest path to the destination.
The algorithm starts from the agent's initial position and explores all reachable nodes, layer by layer, until it reaches the goal. Each node represents a possible position in the environment. This approach guarantees finding the shortest path in unweighted grids where each movement step has an equal cost.
Once the shortest path is computed, the system identifies parcels and delivery points along the route. The agent then follows the path step by step, executing necessary actions such as picking up and delivering parcels.

5.2 PDDL

A PDDL-based planner generates a plan thanks to the interpretation of a given problem description: this is characterized by a sequence of actions that can bring the system from the initial state to the goal state. To achieve this, the planner considers preconditions and effects when exploring potential actions. In this way, it can construct a feasible plan.
In our implementation, we define a fixed *PDDL domain*, specifying the allowable predicates, their parameters, and their associated preconditions and effects. As previously mentioned, PDDL is primarily used for path planning during the movement phase.
To achieve this, the `PddlMove` class is responsible for generating a *PDDL problem* and executing the corresponding actions. Before computing a plan, the agent updates its knowledge of the

environment, ensuring that the most recent map and observed elements, such as other agents and parcels, are considered. A PDDL problem is then dynamically created based on the agent's current position, the goal to be achieved, and the layout of the environment. This problem is passed to an PDDL online solver, which returns a sequence of actions to reach the destination while satisfying the specified constraints.
Once the plan is obtained, the agent follows the computed path step by step, executing movements and interacting with parcels along the way. The implementation also includes mechanisms to handle obstacles, such as other agents blocking the path. When necessary, the agent can coordinate movements with a friendly agent or adapt its delivery strategy if an opponent is in a key location. This ensures efficient path planning and real-time adaptability in a dynamic environment.

5.3 Managing possible intention failure

Both implementations account for the possibility that the agent may not be able to complete the selected action, particularly when it is blocked by another agent. Two specific scenarios are handled:

1. **Universal Case**: When the agent needs to deliver particles but the destination is blocked by an enemy, the agent identifies the second-nearest delivery point and attempts to reach it instead.

2. **Multi-Agent Scenario**: When the agent is blocked by its teammate, the agent closest to the center of the map must move. Further details on this scenario are provided in Section 6.3.

6 Communication

In the multi-agent scenario, effective communication is a very important aspect. Each message consists of a *header*, *content*, and relevant information such as the sender's name and position. Both the *MASTER* and *SLAVE* agents need to continuously exchange messages and share information about their perceived environment to coordinate their actions effectively.

6.1 Handshake

This communication is necessary to establish a connection between *Agent 1 (MASTER)* and *Agent 2 (SLAVE)*. To achieve this, the agents exchange a special handshake message to recognize each other (as in Figure 1).
Initially, the *MASTER* sends a broadcast message with the content `attacchiamo?` to signal the connection request. Upon receiving this message, the *SLAVE* replies with `attacchiamo!` to confirm the recognition.
Once the handshake is complete, both agents set their `friend_id` to establish a mutual connection. After this process, the agents are able to collaborate and proceed with their tasks effectively.

6.2 Environmental information

Agents have the ability to share information about what they perceive. During the detection of parcels and other agents in the map (described respectively in Sec 3.2 and Sec 3.3), `INFO_PARCELS` and `INFO_AGENTS` messages are sent.

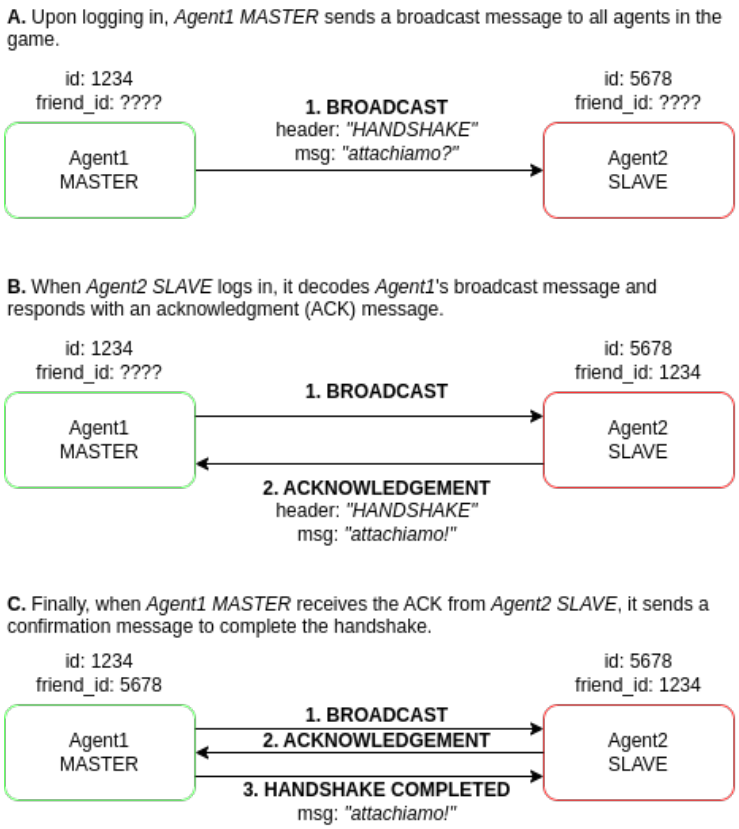

Figure 1: Handshake procedure.

When an agent receives a `INFO_PARCELS` message (sent only if the particles perceived are more than two), it updates its knowledge with the new parcels perceived by its teammate.
The agent then evaluates these parcels based on certain conditions:

- the particle has not yet been collected;
- the particle reward is greater than 4 or the distance from the agent is lower than 20 steps.

If these conditions are met, the agent decides to move towards the parcel to perform the `pick_up` action.

On the other hand, when an agent receives a `INFO_AGENTS` message, it updates its knowledge about the other agents in the environment, adding their position and other relevant information. This helps in maintaining an up-to-date map of agents to facilitate coordination and avoid conflicts.

6.3 Messages of stucked

As mentioned in Sec. 5.3, one agent can block the other during the completion of an intention. In this scenario, the message `STUCKED_TOGETHER` is used.
Before that, the distance to the center of the map of both agents is calculated and ideally, the nearest has to move. Therefore, the content of the message can be either '*You have to move away*' or '*I have to move away*'

If the message contains '*I have to move away*' (as in Figure 2), a confirmation is sent by the other agent. This message will include the content '*You have to move away*'.

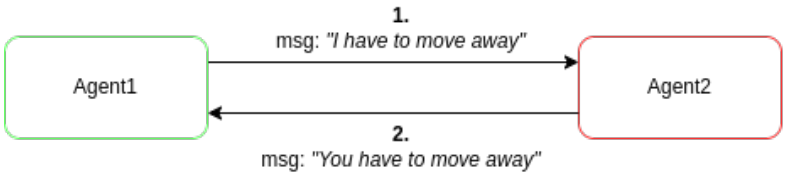

Figure 2: Case *"I have to move away"*.

Once the '*You have to move away*' message is received by an agent (as in Figure 3), it has to calculate its `possible_moves` to determine if it can move. In case of a positive answer (`possible_moves.length > 0`), it drops the packages and moves to a random location, using a simplified version of `moveToRandomPos`. Otherwise, if the agent cannot move, his friend must do so, even though it would be the furthest from the center of the map. In this case, the agent sends a '*You have to move away*' message to instruct the other agent to move.

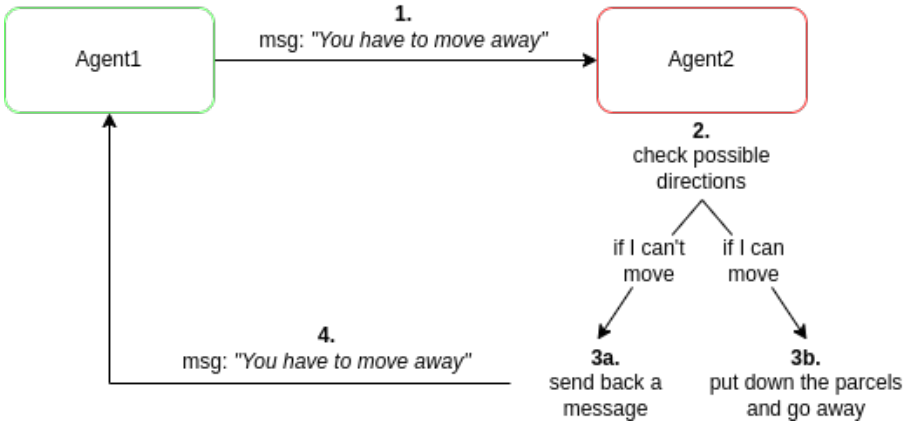

Figure 3: Case *"You have to move away"*.

Below there is a series of images (Figure 4) that illustrate the workflow of the stucked pipeline. Starting from the top-left:

- The *first image* shows the green agent arriving at the delivery point with some particles to drop, but the orange friend agent is already occupying the delivery spot.
- The *second image* shows that the green agent arrives near the orange agent but becomes stuck behind it. Since the green agent is closest to the center of the map, it sends a '*I have to move away*'message. The orange agent responds with '*You have to move away*'. As a result, the green agent drops its particles and moves away.
- The *third image*, shows that the green agent successfully moves away from the stuck position.
- In the *fourth image*, the orange agent senses the particles left by the green agent and proceeds to pick them up.
- Finally, in the *fifth image*, the orange agent successfully delivers the particles, and its score is updated accordingly.

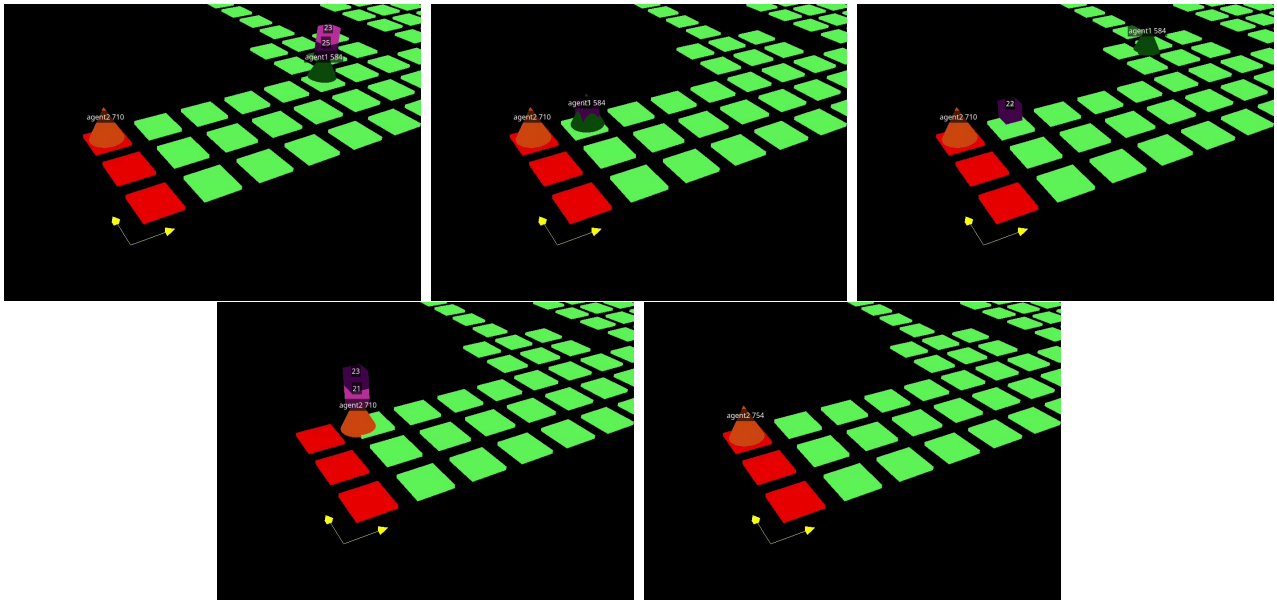

Figure 4: Workflow of the stucked pipeline.

7 Results

To evaluate our agents, we tested them on various maps from the Deliveroo.js server, which provides two sets of levels, each containing nine stages: one for single-agent gameplay and one for multi-agent gameplay. Each test consisted of a 5-minute gameplay session. For planning, we employed a BFS-based strategy, as using PDDL proved impractical due to the latency of the online solver, which requires a new connection every time the agent needs to compute a move.

Tables 1 and 2 present the results for the single-agent and multi-agent scenarios, respectively. Displayed side by side, they allow for a direct comparison of performance across different levels.

Level	Score
24c1_1	1220
24c1_2	758
24c1_3	1603
24c1_4	148
24c1_5	6941
24c1_6	443
24c1_7	586
24c1_8	631
24c1_9	778

Table 1: Results for Single-Agent

Level	Total score	A1 score	A2 score
24c2_1	1640	700	940
24c2_2	4343	2203	2140
24c2_3	2657	841	1816
24c2_4	1849	1346	503
24c2_5	1297	603	694
24c2_6	14908	8259	6656
24c2_7	6986	3454	3532
24c2_8	320	155	165
24c2_9	5204	2318	2886

Table 2: Results for Multi-Agent

8 Conclusion and future improvements

In this project, an autonomous agent system has been developed. It can succeed in the challenges of parcel collection and delivery within a dynamic environment. Using the Belief-Desire-Intention (BDI) architecture, the agents demonstrated strong capabilities in making decisions,

adjusting their strategies, and coordinating with each other. Communication between agents is efficient, particularly in resolving situations where one agent is stucked with another. The agents also continuously update their knowledge base, enabling informed decision-making and smooth coordination.
Our evaluation on different maps confirmed that the agents effectively adapted to various challenges. The BFS-based planning approach proved to be the most reliable and efficient method for path-finding, enabling the agents to quickly find paths even in dynamic environments. The PDDL-based planner, though conceptually promising, was limited by its execution latency, making it less suitable for fast real-time decision-making.
Additionally, in the single-agent scenario, our approach performed better on maps with randomized spawning cells, allowing the agent to adapt more effectively to the dynamic environment. However, it performed slightly worse on maps with fixed spawning points, especially when those points were located in the corners of the map (e.g., level `24c1_4`).

For future improvements, we aim to enhance adaptability by refining coordination mechanisms in multi-agent scenarios, optimizing distance calculations using `value_coords`, and exploring alternative planning methods that balance efficiency and accuracy. Additionally, integrating learning-based approaches could allow agents to dynamically adjust their strategies over time. With these enhancements, our autonomous system could become even more robust and effective in real-world applications.