BDI Agent for Deliveroo Game

Panini Martina 257851 martina.panini@studenti.unitn.it
Avanzolini Elia 256153 elia.avanzolini@studenti.unitn.it

June 10, 2025

Abstract

This report details the design and implementation of a BDI (Belief-Desire-Intention) agent system designed for the Deliveroo game. First a single-agent was implemented and then a multi-agent architecture. We describe the core BDI architecture, how beliefs, options, and intentions are managed, the planning process, and how coordination and communication are handled in the multi-agent extension.

Contents

1 Introduction

In this report will be discussed the architecture of the project for the "Autonomous Software Agents" course, based on the Deliveroo game. The objective of the game is to earn as many points as possible. This goal can be reached by picking up parcels and then deliver them in the designed delivery zones. This has be done avoiding other agents and not accessible tiles, these actions lead to collect penalties.
The aim of the project is to implement a BDI autonomous agent. The BDI architecture allows the agent to perceive the environment, establish goal and then select the plan to achieve that goal. These actions are done continuously in a loop.
The project is divided into two main parts, reflecting the two modes of the Deliveroo game.

Single Agent: build a single autonomous agent capable of understanding and processing information from its environment.

Multi Agent: add a second autonomous agent. The agents cooperate by sharing information about their respective beliefs and goals.

The evaluation of performance of both parts is done mainly across two challenges against other agents.

2 Beliefs and Beliefs Revision

Beliefs are the agent's perceptions of its environment. In this case, he has to keep track of the parcels and other agents he sees. In addition, beliefs include the current state of the agent and of the map.

2.1 Parcels

Parcel detection is handled by the asynchronous function `onParcelsSensing`, which is triggered whenever a parcel is detected. For each detected parcel, we record its location (x and y coordinates) and reward value.
A control is made to ensure that only the parcels within the observation distance are perceived.
In the multi-agent mode this perception is shared with the teammate.

2.2 Other agents

The asynchronous function `onAgentsSensing` is responsible for updating the information about opposing agents detected by the agent.
Every time an agent is perceived, he will be added to a list to keep track of the agents seen.
In the multi-agent mode this perception is shared with the teammate.

2.2.1 Other agents collision

When the agent collides with others, he collects penalties. To avoid this, before moving, agent checks the list with other agents sensed and rejects paths that will lead against other agents. This is done before every move. Unfortunately, this is not sufficient to avoid completely the collision with other agents due to the difference between the velocity of the movement and the velocity of take decisions. This vulnerability, as we have seen during

challenges, causes a small penalty increase with respect to the points earned. So we have chosen to maintain a simpler collision control and ensure a fast movement, rather than implement a more complex control.

2.3 Map management

The agent keeps an internal model of the map, which it updates every time new information arrives from the environment. This model stores this information in a structured way using a map object that keeps track of all tiles and their types (such as walls, spawn points, or delivery points). Have a knowledge about tiles type helps agent to find paths avoiding unavailable tiles.

This set of beliefs is regularly rebuilt using the latest map information, ensuring that the agent's understanding of the environment is always up to date.

3 Intentions and Intentions Revision

Beliefs are elaborated to understand what is the next move. Agent create a sorted list of options that include `pick_up` and `go_deliver`. Best option is pushed in an intention queue and the agent can choose what to do next.

3.1 Options

The options generation function is the core of the agent's decision-making process. It evaluates various potential actions (options), assigns a utility to each, and then selects the option with the highest utility to be executed. The agent's decision-making is influenced by its current state (beliefs), its goals (delivering parcels), and its interactions with the environment and, eventually, a teammate.

The process begins with the agent "clearing out" previous decisions, preparing to consider a new set of possibilities.

If the agent is already carrying parcels he compute the utility of delivery based on his distance to the delivery tile and on the reward given by the parcels he is carrying. This utility must be continuously computed because the reward of parcels decrease with time. Moreover, if its teammate has already expressed an intention to deliver to the same point, the agent decrease its utility to deliver, to avoid unnecessarily duplicating efforts.

If he's not carrying anything, or even while considering delivery, the agent evaluates pickup opportunities. It scans the environment to locate nearby parcels, but not just any parcel. It won't consider those not available, or those its teammate already intends to pick up. For each eligible parcel, it calculates the utility of taking it, considering both its immediate value and the total time required to pick it up and then deliver it. Here too, the agent communicates its pickup intentions to its teammate, ensuring that both don't head for the same parcel, thus optimizing team efficiency.

Once the agent has considered all possible options it sorts them based on their estimated utility. The option with the highest utility becomes its intention and is queued for execution. If, for some reason, no valid options are generated (e.g., no parcels to pick up, no parcels to deliver, or all perceived parcels are blocked/carried), the agent decides to explore the environment. This cycle repeats continuously, allowing the agent to dynamically adapt to the environment and achieve its goals as efficiently as possible.

3.1.1 Utility functions

Utility is essentially a score that indicates the best option for the agent in that moment: the option with the higher score is chosen.

Delivery option: The delivery utility is computed using the following formula that balances value and cost over time:

$$util = carriedReward - (carriedQty * \frac{movement\_duration}{parcel\_decading\_interval} * d2d) * deliveryPenalty;$$

- `carriedReward`: The total reward of the parcels currently carried by the agent.
- `carriedQty`: The number of parcels the agent is currently carrying.
- `movement_duration`: The estimated time the agent needs to make a movement. It is given by configuration file.
- `parcel_decaying_interval`: The time interval after which each parcel's value decreases.
- `d2d`: Measure the distance between the agent and the delivery zone. It scales the loss depending on distance.
- `deliveryPenalty`: A penalty factor applied if a teammate has already committed to the same delivery, discouraging redundant efforts and encouraging task allocation.

Pickup option: To decide if picking up a particular parcel, the agent calculates its overall utility. The agent adds the reward of the new parcel to the total reward of all the parcels it's already carrying. From this combined potential earning, it then subtracts the decay penalty (time to grab the parcel plus time to delivery it), reflecting how much value might be lost along the way. In this case he didn't aplly a penalty to the parcels that the team mate want to pick because they are directly discarded.

$$util = (carriedReward + parcel\_reward - decayPenalty);$$

- `decayPenalty`: The estimated total loss in parcel value due to delivery time and current load.
- `carriedReward`: The current total reward value of the parcels already being carried.
- `parcel_reward`: The reward value of the parcel the agent is considering picking up.

`decayPenalty` and `carriedReward` are calculated with dedicated functions.

3.2 Intentions

The intention management in the BDI agent is handled through a dedicated loop implemented in the `IntentionRevision` class and its subclass IntentionRevisionReplace. This mechanism ensures that the agent constantly evaluates and executes its current intentions, adapting dynamically to changes in the environment and available options.

At the core of the system is the `loop()` method, which continuously checks whether there is any intention currently in the queue. If there is, the agent attempts to achieve the first intention by executing a corresponding plan from the plan library. Each intention encapsulates a predicate that defines a specific goal or action to be achieved. When the

agent attempts to achieve an intention, it iterates through its available plans to find one that is applicable to the given predicate. Once found, the plan is executed. If execution is successful, the intention is removed from the queue. If it fails either because no suitable plan was found or because the plan execution failed the intention is still removed, and the system takes additional steps to adapt: it removes the failed option from the options list and, if the failed action involved picking up a parcel, marks the parcel as blocked to avoid retrying it immediately. After each failure, the system also regenerates the set of available options, keeping the decision-making process up-to-date.

The subclass `IntentionRevisionReplace` adds a mechanism to manage intention replacement. When a new predicate is pushed, the agent compares the utility of the new intention with the one currently at the front of the queue. If the current intention is a general exploration task or if the new intention has higher utility, the existing intention is stopped and replaced. This allows the agent to act opportunistically and prioritize more beneficial actions as they become available, rather than sticking to a rigid execution order.

4 Planning

The planning phase of the BDI agent is centered around procedures that define how specific intentions are to be achieved. Each plan is associated with a predicate that describes a certain type of action. The agent selects a plan dynamically based on whether it is applicable to the current intention.

At the core, each plan encapsulates a strategy to achieve a particular subgoal. This structure supports hierarchical planning, where high-level intentions can be broken down into a sequence of smaller, coordinated sub-intentions.

Moving from a position to another, which is central to most goals, is handled by the `AStarMove` plan. It employs an A^* pathfinding algorithm to find a viable route to a given destination, while dynamically avoiding walls, agents, and teammates. The movement is reactive: if the path becomes blocked, or if the agent is interrupted (e.g., stopped externally), the plan fails and the agent re-generate options.

We chose A^* search because it finds the shortest path efficiently by combining actual distance and a heuristic, making it fast and reliable for dynamic navigation.

The agent supports autonomous and exploratory behaviors, to allow movement even if he has no options available. `ExploreSpawnTiles`, `RandomMove` and `SmartExplore` arethe classes for the exploration.

- *RandomMove* is the simplest way to move the agent toward the map. He simply choose the next available tile randomly. Using this plan he covers a little area.
- *SmartExplore* uses a scoring heuristic to move toward less-visited or more promising areas of the map, balancing exploration efficiency and coverage.
- *ExploreSpawnTiles* continuously guides the agent through known spawn locations. This plan is useful in maps where there are few and sparse spawn tiles. In these cases, if we use SmartExplore, the agent risks to not reach spawn tiles during his exploration. These considerations were made after the first challenge, where our agent performed poorly in maps with few spawn tiles.

For each plan, tiles are filtered checking if they are walls or are already occupied by other agents. This allow the agent to go towards only available tiles without collecting penalties.

Each plan can be stopped at any point, enabling the system to switch strategies or abandon a goal if conditions change.

4.1 PDDL

In the single-agent scenario, the path to reach a target, is computed by PDDL planner. By describing the world in terms of predicates and actions, the agent can dynamically generate plans to achieve specific goals. This approach offers flexibility making it easier to handle complex tasks.
PDDL planner is not so reactive, so it is not the optimal choice for the multi-agent scenario where the agent have to revise his intentions every time he receive a message from his team mate.
For this reason we have decide to use it only in the single-agent scenario where the agent have to decide only on the basis on his own beliefs.

5 Multi-Agent communication

In the multi-agent configuration, the system launches two distinct agent processes. Each agent is also made aware of its teammate's ID during startup.

Agents share messages about their beliefs, allowing them to inform their teammates about currently visible parcels and other agents. The core of the communication is the sharing of their own intentions. This targeted communication aims to prevent redundant efforts and optimize task allocation.

The primary coordination mechanism revolves around two specific message types: `pickup_intentions` and `delivery_intention`.

- **Pickup Intentions:** When an agent decides to pick up certain parcels, it communicates this intent to its teammate by sending a message. This message contains an array of parcel IDs that the sending agent plans to collect. Upon receiving this message, the teammate updates a local set, which keeps track of all parcels currently targeted by the other agent. This shared knowledge helps agents avoid pursuing the same parcel simultaneously, allowing them to distribute the pickup tasks more effectively.

- **Delivery Intentions:** Similarly, when an agent intends to make a delivery, he sends a message. This message provides details about the delivery, including the target coordinates. The receiving agent stores this information and apply a penalty for that delivery zone. In this case, the delivery zone chosen by the team mate is not directly discarded because in the meantime the agent reaches the zone, the teammate has most likely already moved away.

This exchange of intentions allows the agents to build a dynamic, shared understanding of each other's current goals. By knowing which parcels its teammate is picking up and where it intends to deliver, an agent can make more informed decisions about its own actions.

The figure 1 shows how this logic works.

6 Conclusions & Benchmarks

This project demonstrated the design and implementation of a BDI-based agent capable of operating autonomously in the Deliveroo game, both in single and multi-agent modes. The agent continuously updates its beliefs, evaluates options through utility functions, and selects the most effective intentions to pursue.

In the multi-agent setting, coordination was achieved through the exchange of pickup and delivery intentions. This reduced task overlap and improved overall efficiency. Utility functions played a key role by factoring in parcel value decay, travel time, and teammate actions, allowing the agents to balance individual performance with team collaboration.

The architecture proved flexible and reactive, with dynamic plan selection and exploration behaviors enabling adaptation to changing environments. However, some simplifications, such as basic collision avoidance, led to occasional penalties, showing a trade-off between decision speed and control accuracy.

Future improvements could include:

- Smarter collision prediction using movement history.
- Adaptive utility functions that learn from past successes and failures.
- A more structured task allocation mechanism in the multi-agent mode.
- Improved exploration strategies based on map coverage analysis.
- Improve coordination between agents in a more complex environment. At the moment the two agents simply share intentions with the team mate, but in scenario where is necessary collaboration (e.g. hallway scenario) they fail.

Overall, the system effectively leverages the BDI model to create intelligent and coordinated agents, as demonstrated by the results presented in Tables 1 and 2 from the two challenges done against other agents and teams. This foundational success shows strong potential for further enhancement in both robustness and collaboration.

Q1				Q2				Q3			
1	2	3	Final	pts	pts	pts	Final	pts	pts	pts	Final
13	8	14	35	7	6	4	17	2	4	1	7

Table 1: Results of Challenge 1: 4^{th} placement

1	2	3	4	5	6	7	8	9	Final
156	259	613	436	84	66	617	252	275	2758

Table 2: Results of Challenge 2: 3^{rd} placement

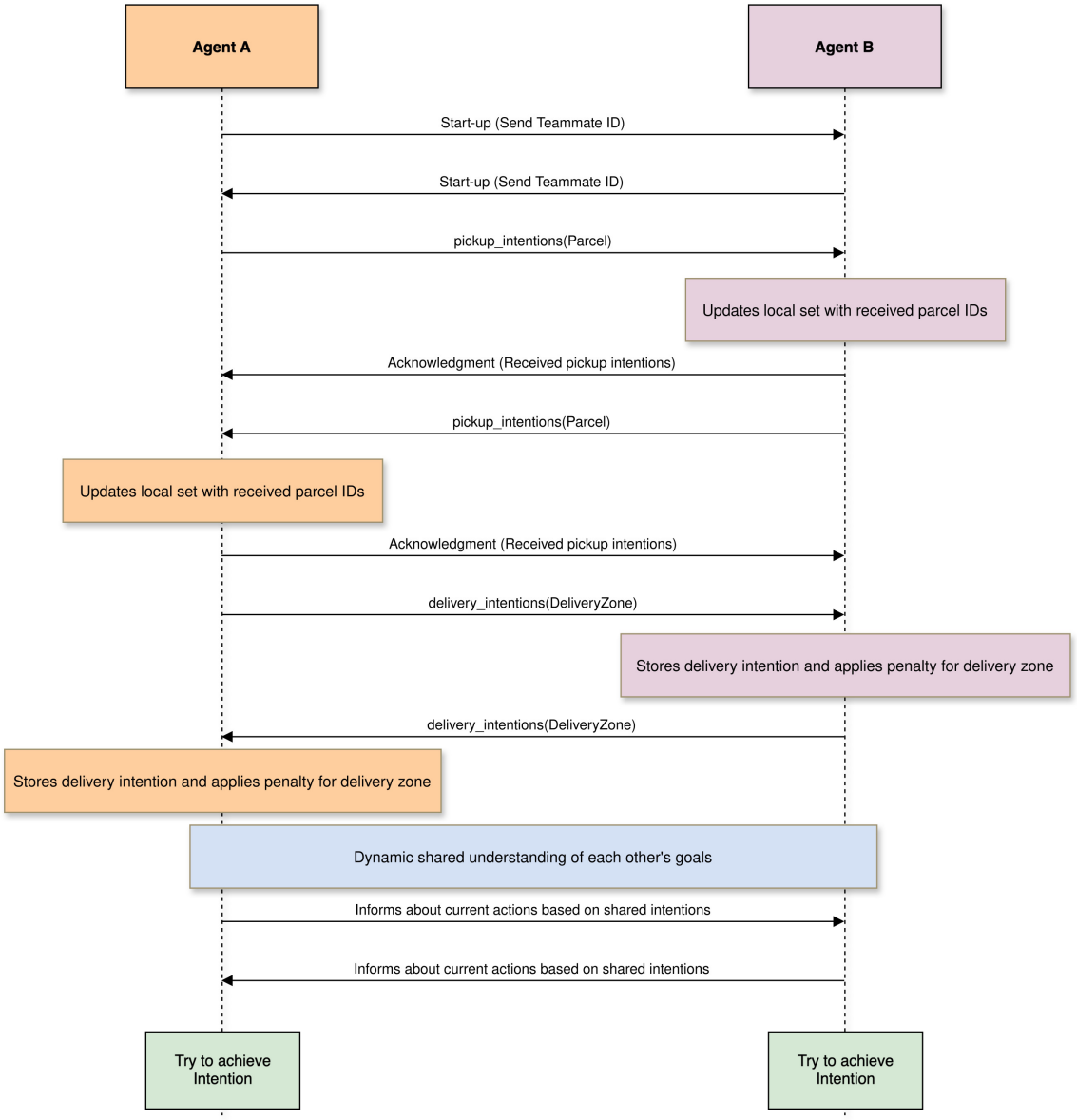

Figure 1: Scheme to visualize the coordination in multi agent mode