

Autonomous Software Agents Project Report

Cappelletti Samuele - 247367 Lazzerini Thomas - 247368
samuele.cappelletti@studenti.unitn.it thomas.lazzerini@studenti.unitn.it

July 29, 2025

1 Introduction

This document contains the final report for the Autonomous Software Agents project, where one or more agents have to play in the Deliveroo.js game. Agents developed in this project must implement the BDI (*Beliefs, Desires, Intentions*) architecture (Fig. 1) where the agent is able to sense the environment defining an internal belief, generate a set of possible intentions, and commit to one or multiple of them via a plan-based system while performing a constant revision of both intentions and beliefs.

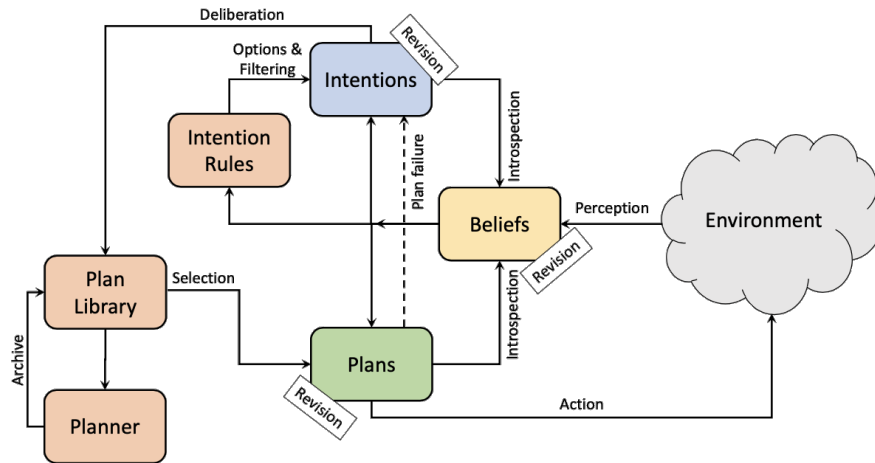


Figure 1: BDI architecture diagram used during the agent development

The project is composed by a single agent part and a multi agent part. The first part should include a single agent implementing the basic functions necessary to him to work correctly. In particular, the agent should represents and mange beliefs from sensing data activate intentions and act on the environment and use predefined plans to achieve its intentions. All of this, while performing a constant revision of beliefs and intentions to allow him to stop, hold or invalid the current running intention. This permits the agent to act accordingly in a rapid evolving environment. Once these functions have been achieved, the agent should interact with an automated planning utility to get the plan of actions to perform.

The second part should introduce a second agent able to cooperate with the first one, and viceversa, to achieve the goal. In particular, the two agents should be able to communicate, exchange beliefs, coordinate and negotiate possible solutions to achieve the goal, which may not be achievable by a single agent.

Furthermore, both should be able to operate in different scenarios, involving other competitive agents and rapidly evolving environments.

2 Single Agent

This part regards the development of a simple single agent, able to perform the basic functions to operate in the *Deliveroo.js* setting. In particular, this should include the ability to represents and manage an internal belief system, built from sensing data received from the server and able to perform revision of outdated or no longer valid beliefs. Based on these beliefs, the agent should be able to define and activate intentions, also performing revision of older intentions, and also act in the environment to achieve such intentions. To achieve the intentions the agent should use a set of predefined plans or an integration with a planner utility to perform the correct actions.

In particular, our single agent operates with the same script as the multi agent configuration, as this multi agent configuration is resilient to the absence of the second agent. Specifically, the single agent configuration will avoid to send messages to the non existing companion and, in all of those situations where it is necessary to know the companion information, like its position or the parcels he is carrying, the agent will consider the companion as null, so like he is nowhere and he is carrying no parcels.

2.1 Initial Connection

The first step is to initialize the agent's *belief set*, this include the initialization of an empty memory. Then, the script will evaluate the command line parameters and will initialize the connection using a specific single agent token or, if required, a token created on the fly. The script will initialize the intention system, will add the predefined plans to its internal library and will initialize the callbacks for the *updates* from the server, like "*onParcelsSensing*" and "*onYou*". Lastly, the script defines a *Promise* on the "*onMap*" and "*onConfig*" callbacks on their resolution before continuing with the normal execution, in particular a temporized loop both on the "*agent's memory revision*" and the "*option generation*" phases.

2.2 Belief

In this part we cover the *BeliefSet* class regarding the single agent part. In particular it is composed by an initial setup phase where we define the game map and the game configurations, then an "*environment sensing*" part where we update the agent's belief using the information sensed from the environment and a last "*memory revision*" phase where we remove old and unreliable information from the memory.

2.2.1 Initial Configuration

The first step includes the initialization of the internal memory, the most important components are the "*agent memory*", in which is stored the information about the other agents, the "*parcel memory*", in which is stored the information about the parcels, and the "*me memory*", in which is stored the information about our agent, like its position, id and token.

The second step is to define an internal representation of the game map. Firstly, we define a simple matrix containing the type of the tiles given their $[x, y]$ position. Then we perform a check on the validity of each cell, automatically setting as "wall" (type 0) all those cells that have no neighbors and thus not reachable. Lastly, we use the information of this matrix to define a graph structure on the map itself, where each node represent a single tile containing the associated position, type and neighbors. This allow us to perform efficient navigation using BFS search. Associated to this map, we also define another matrix called "*time map*", of the same sizes of the game map, which contain the "last visited timestamp" of each map tile. This map is needed during the exploration phase.

Finally, we save the game configuration in a dedicated map, to have an easier access to it during the game execution.

2.2.2 Environment Sensing

During the normal execution, the agent will receive the information about the parcels and the agents that he is able to sense and its own updated information from the server. Upon reception of these messages, the belief set will update its internal representation, to be coherent with the new setting of the environment. In particular, the *"onYou"* handler will update the agent's id (necessary in the case in which we generate a random token on the fly), its current position and the time map, accordingly to the current agent's position (all the cells in the agent field of view are updated with the current timestamp).

Regarding the *"onParcelsSensing"*, the server will send the information of the parcels sensed by the agent, which will be inserted in its internal parcel memory updating the entries of already seen parcels and creating new entries for the newly seen parcels. Furthermore, the timestamp of the sensing is inserted in the entry, to allow a correct revision phase later on. The same process is performed also for the *"onAgentsSensing"*.

2.2.3 Memory Revision

The agent's belief may contain old information, like a parcel that is no longer visible because out of the sensing range, which may be relevant for some time but, if it become too old, it may be unreliable. For this reason, the last fundamental part of the *BeliefSet* class is the *memory revision* component, which goal is to remove this old and unreliable information. Another important notion of the *Deliveroo.js* setting is that the server provide the sensing information only when something happens, like a parcel appears or an agent enters the agent sensing range, but not the contrary: if a parcel expires or is not there anymore, or an agent moves outside of the sensing range, then the server will not notify the agent. For this reason, we decided to adopt a *time window based approach*.

Regarding the parcel's revision, we cycle all the parcels in the memory. If the position of the parcel falls into the agent's sensing range and the last time we saw the parcel is recent (less than an *agent's movement duration*, which, generally, is the shortest time period in which something may happen both to a parcel and to an agent) then we can safely keep the parcel as we know it is still there, otherwise we know that the parcel isn't there anymore (because it has expired or it has been picked up by another agent) so we can remove it from the memory. Otherwise, if the parcel is not in the agent's sensing range, we consider a larger time window, since we can't be sure if the parcel is still there or not, before removing it from the memory. The same approach is considered also for the *agent memory*.

2.3 Options and Filtering

To allow the agent to act in the environment, it must first identify possible and desirable options given the current state of his beliefs. Once all the options have been identified, a filtering operation has to be performed to select the best one.

To do so, we implemented an *optionsGeneration* function. This will first identify the best option (i.e. with highest reward, refer to 2.3.1 for the reward definition) between the best *"go_pick_up"* option and the *"go_deliver"* option to the nearest free delivery cell considering the current agent's beliefs. Then, the best option is compared to the currently executed intention and, if it is better, the current intention is stopped and replaced with the new option. In the function are also present several controls to prioritize the completion of *"go_pick_up"* options in some specific conditions. In particular, we identified two conditions where the currently executed intention is a *"go_pick_up"* and we want to let it finish: the first one is when there is no best option, in this case we let the current intention finish to avoid pushing an *"explore"* option. While, the second one is when the best option is a *"go_deliver"*, in this case we let the current intention finish to avoid unwanted behaviors, like an agent that is moving to pick up a parcel and changes direction to go deliver the parcels when he is close to the parcel to pickup. Moreover, we also identified

another specific condition where there is no best option and the current intention is not *"go_pick_up"*. In this case, we generate an *"explore"* option randomly choosing between a *"time-based"* or a *"distance-based"* strategy, giving more weight to the *"time-based"*. Additionally, in the *optionsGeneration* function we set a flag to avoid executing it multiple times in parallel, and we also perform a check to avoid executing it while generating a plan using an automated planner.

2.3.1 Filtering the Best *"go_pick_up"* Option

For each parcel in the agent's memory that is not carried by anyone (i.e., *free parcel*), we generate a *"go_pick_up"* option and the associated estimated reward. Such estimation considers both the reward from the parcel to pick up and the reward expected from the currently carried parcels. To do so, we consider both the total path, computed via BFS, to go pick up and go deliver the parcel to the nearest reachable delivery, the agent's movement duration and parcel decay interval to project the parcels' reward in time and the last visit time for the parcel to pick up to predict the current score even out of the sensing range. The reward obtained is then scaled by several fixed factors to give more weight to the nearest parcels. This behavior is justified by the idea that picking up a near low reward parcel is cheaper than picking up a distant high reward parcel.

Once all the options are ready, we select the one with the highest reward, and, if multiple options have the same reward, we discriminate on the distance from the agent.

If no best option is selected at this point, it means that all the options have reward zero, most probably because all the deliveries are occupied by other agents. In this case we should still select the best option in a meaningful way. To do so, we compute a simple version of the expected reward considering the ratio between the parcel's *"raw"* score and its distance from the agent.

2.3.2 Computing *"go_deliver"* Option

To construct the *"go_deliver"* option to the nearest reachable delivery, the first step is to check if the agent is carrying parcels: if not, there is no need for a *"go_deliver"* option. Otherwise, we compute the path to reach the nearest delivery cell and use such path to compute the expected reward of the carried parcels similarly to the best *"go_pick_up"* option (2.3.1). If such path does not exist, it means that other agents are blocking all the deliveries, therefore we can not generate a *"go_deliver"* option.

To avoid unwanted behaviors, like a dominance of *"go_pick_up"* options, we track the number of moves performed by the agent while carrying parcels. We use this counter, paired with a fixed constant, to scale up the reward of the *"go_deliver"* option. This will increase the probability of performing a delivery as the moves increase. After performing the delivery, the moves counter is reset.

2.4 Intention

The intention's management part is highly inspired by the code presented in the laboratories with some modifications. The most important regards the *"intention revision"*, in particular we removed the intention queue, considering only the current and next intentions. We decided to adopt this approach considering the rapidly-evolving nature of the Deliveroo.js environment, replacing a queue of multiple sequential intention that probably will be invalid when executed, in favor of a simpler system that consider only the current intention, which is being achieved, and a possible next intention, which the agent consider better than the current intention and will replace it as soon as possible.

The *async loop* function in the *"IntentionRevisionReplace"* class, serves the current intention, if existing. Once this intention is achieved, it is removed as current intention and replaced by the next intention, if existing. On the other hand, the *async push* function in the same class, will receive a new option from the

optionsGeneration and shall decide if saving it as the next intention. In this case, the *stop* function will be called on the current intention, causing it to stop as soon as possible, allowing the *loop* function to replace it with the next intention.

During the *optionsGeneration*, it may happen that the same option is pushed multiple times rapidly, for this reason, there is the need to understand if the next intention is really a new intention, so it must be replaced, or just a copy of the same intention, that must be discarded. To do so, we implemented a comparison function (*areIntentionsEqual*), that compares the option's parameters with the intention's predicate.

2.5 Plan

spiegare i vari piani, come li gestiamo, come facciamo revision dei piani (come la mettiamo insieme a option generation)

2.6 Planning

come abbiamo implementato il planning e per cosa, spiegare che fa schifo, spiegare roba dei negative prepositions (in conflitto con closed world assumption) che fa crashare il FF planner

2.7 Strategy

TODO vedere se mettere o cosa perche tempi option generation e memory revision

3 Multi Agent

PICCOLA INTRODUZIONE PICCOLA INTRODUZIONE PER DIRE COSA È STATO INTRODOTTO
mettere che il file è unico e lanciamo single o multi agent con parametri (single agent è come il multi agent ma senza pal, agente funziona bene anche senza bisogno di pal (possiamo aggiungere e togliere il secondo agente in qualsiasi momento e il primo continua a funzionare))

3.1 Multi Agent Memory

spiegare cosa abbiamo aggiunto nel belief del single agent

3.2 Communication

spiegare i vari messaggi e il tipo di comunicazione che usiamo

3.3 Belief

spiegare come aggiorniamo belief con info ricevute da pal

3.4 Options and Filtering

spiegare calcolo reward anche per il pal per scegliere solo le options che ci convengono, spiegare come aggiungiamo la option per scambiare parcel (counter per evitare scambi non voluti)

3.5 Plans

spiegare piani aggiunti per scambio carried parcels