# Autonomous Software Agents Project Report

Cappelletti Samuele - 247367
samuele.cappelletti@studenti.unitn.it

Lazzerini Thomas - 247368
thomas.lazzerini@studenti.unitn.it

July 30, 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.

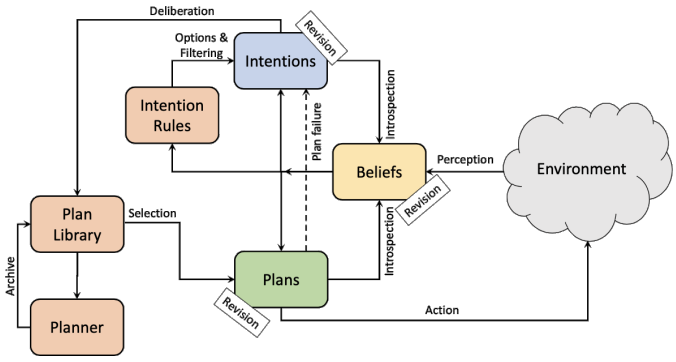


*Figure 1: BDI architecture diagram used during the agent development*

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. The temporization of both these loops is set, at the moment, at 50 *ms*: from our tests, this is the shortest possible time period in which something significant may happen, the lower bound being the agent movement duration. This decision is justified by the rapidly-evolving nature of the Deliveroo.js environment, thus keeping the overall approach simple and avoid manually calling these function.

## 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

Due to the fact that we divided the code in multiple files, it was necessary to share somehow the plans, defined in the *agent.js* file. For this reason, we created the plan library directly in the "*IntentionRevision*" class and add the plans directly there. Then, when calling the *achieve* function on the intention we provide as parameter the plan library, needed select the correct plan to achieve the intention. Similarly, the basic "*Plan*" class is the same as

the one presented in the laboratories with the difference that we provide the plan library when calling a sub-intention.

Using the "*Plan*" class we defined the list of predefined plans used by the agent to execute the different intentions.

### 2.5.1 Explore Plan

The "*Explore*" plan is used to serve the "*explore*" intention. It is composed by a "*go_to*" sub-intention with the goal to explore the map and discover parcels to pickup. The coordinates to reach are computed using different randomly selected strategies: a time-based or a distance-based. Both of them compute at first a list of explorable cells using BFS. Then, the former will use the *time map* defined in the "*BeliefSet*" to incentivize those positions that we have not seen in a long time. On the other hand, the latter will simply select distant cells with higher probability. If, for some reason, we do not find any valid cell in the *timed explore*, we automatically perform a *distance explore*.

The probability of the cells to explore in the time-based strategy is computed as follows: the first step is to compute the *age* of all the explorable cells (i.e., the ms we have not seen that cell) and normalize them in a range between $[0, 1]$. In the meantime, we also compute the *Manhattan distance* of each cell from the agent's position, and also apply a logarithm of the $distance + 1$ to adjust the scale between age and distance. After this, we compute a softmax-like *priority* of each cell as:

$$priority_{cell} = \frac{e^{\alpha \cdot age_{cell} - \beta \cdot distance_{cell}}}{\sum_{c \in cells} e^{\alpha \cdot age_c - \beta \cdot distance_c}}$$

From our testings, we found out the best values for $\alpha$ and $\beta$ to be respectively 5 and 1. After this, we simply select randomly a cell using the associated priority as probability.

### 2.5.2 GoPickUp Plan

The "*GoPickUp*" plan is used to serve the "*go_pick_up*" intention. It is simply composed by a "*go_to*" sub-intention to move to the desired cell where the parcel is located and a pickup action performed in a simple wrapper that avoids multiple emit actions in parallel, manually save the picked up parcel as carried by me and perform the pickup action.

### 2.5.3 GoDeliver Plan

The "*GoDeliver*" plan is used to serve the "*go_deliver*" intention. Similarly to the "*GoPickUp*" plan, it is composed by a "*go_to*" sub-intention to move to the desired delivery cell and a putdown action performed in a wrapper that avoids multiple emit actions in parallel, resets the moves counter, manually removes the carried parcels (and adds them to an ignore list to avoid consistency problems) and performs the putdown action.

### 2.5.4 Move Plan

The "*Move*" plan is used to serve the "*go_to*" intention. The first step is to compte a path from the agent's current position to the target coordinates using BFS or an automated planner (detailed in 2.6). Once the path is defined (if not, the plan fails) the agent will follow it move by move until it reaches the final position, where the plan terminates with success.

During the path-following phase, the agent performs a set of checks to ensure the validity and relevance of the action. In particular, the first regards the "*go_pick_up*" intentions, where the agent checks if at the final position there is still a parcel to pick up, if not the intention fails because it is no longer relevant. Another check regards the path itself, as it may happen that another agent blocks a cell in the path. For this reason, the agent maintains an internal counter: if the following path results to be occupied by anyone for more than three consecutive moves, then the path is considered no longer valid and the intention fails. Similarly, another check regards the adjacent cell in which the agent wants to move to, and if that cell appears to be occupied, then the intention fails immediately. Lastly, a final check is a fail-safe to manage the case in which, for some reason, the agent was unable to move and got stuck. Also in this case, the intention is failed.

In all of these cases, the intention fails regardless of the situation. This respects our strategy of single and frequent option generation: instead of trying to fix an invalid intention, we make it fail and, almost

immediately, another one, which we are sure to be valid and relevant, is ready to take its place.

## 2.6 Planning

In the domain of autonomous software agents, plans can be automatically generated using *automated planning*, typically with the aid of the *PDDL language* and various generic planners. These planners take as input a *domain file*, which defines the entities, actions, and rules of the domain, and a *problem file*, which specifies a particular instance, including the environment, agents, and the goal to achieve.

### 2.6.1 Domain Definition

We define our domain in a dedicated domain file, in particular we decided to model the "*go_to*" intention using *automated planning* to obtain the associated path. For this reason, the domain models the required components to achieve the move, discarding other components like the parcels and their scores.

In particular, to define the map itself we use a set of *tile* entities that are connected among them using the predicates *right*, *left*, *up* and *down* (e.g., the predicate "*right ?t1 ?t2*" means that the tile *?t1* is right of *?t2*). Furthermore, we use the *free* predicate to identify those tiles that are not occupied by any agent. Finally, we also use a *me* predicate to identify the agent itself and an *at* predicate to identify the tile in which the agent itself is located.

Regarding the actions, we defined to ones associated to the move itself, in particular *right*, *left*, *up* and *down* (notice that the actions are different from the former predicates despite the same name). We take as example the *right* action, then the other ones are extremely similar. As parameters we consider the agent's representation, the initial tile in which the agent is located and the next tile in which the agent wants to move. The first step is to consider the *preconditions*: the agent must be the actual agent, he must be located at the initial tile, the next tile must be right with respect to the initial tile and the next tile must be free. If the preconditions are men, we can apply the effects, in particular: the agent is now located at the next tile and no longer at the initial tile, consequently the initial tile becomes free and the next tile becomes no longer free.

### 2.6.2 Problem Definition

Differently from the domain, the problem has to be defined on the fly because the environment changes constantly. For this reason, we decided to include the problem definition directly into the agent's belief set, using the planning dedicated classes presented in the laboratories.

The first step is to instantiate a new "*Beliefset*" class (which is different from our "*BeliefSet*" class) during the agent's belief constructor. Then, when the agent receives the map information from the server, he will build its internal map representation and then he will use it to create its planning counterpart. In particular, we represent the specific tile as "*x<x coord>y<y coord>*", instantiate all the *tiles* entities (only for those walkable cells), initially set them as *free* and connect them to their neighbors using the direction predicates.

Once the initial configuration is complete, we have to update the planning belief as the environment evolves and the agent senses it. In particular, we have to update the agent's current position by invalidation the older *at* predicate and declaring the old cell as *free*. Consequently, we remove the *free* predicate for the new cell and also declare the new *at* predicate for it. We do this procedure after every *onYou* update from the server and, to be sure to always have updated information, also after every move during the *Move* plan. Similarly, we have also to update the other agents' positions by applying and removing the *free* predicate both in the *onAgentsSensing* handler and in the *reviseMemory* function.

At this point, the planning belief set is updated and ready to generate a problem file using the same approach shown in the laboratories. To do so, we defined a dedicated function that takes as input the coordinates of the destination cell of the "*go_to*" intention and automatically sets as goal of the problem the agent being *at* that position. After this, we apply some post processing on the problem file to fix an issue we identified during our testings. In particular, the "*Beliefset*" class shown in the laboratories, when computing an undeclare, instead of just removing the positive atoms it also adds a negative counterpart, which is inconsistent with the *closed world assumption* and, besides being semantically

wrong, it causes the FF planner to crash when a negative atom is inserted in the last position of the problem init. To fix this, we applied a regular expression to remove all the negative atoms.

### 2.6.3 Applying the Automated Planner

As said before, we decided to apply the automated planning to compute the navigation path in the *Move* plan. Once we defined the problem's string, we can feed it and the domain to the *onlineSolver* function presented in the laboratories. This will return a plan containing the sequence of actions to go from the agent's current position to the desired position. From this plan, we simply cycle all the actions and convert them in the same format as the ones we get from the BFS search (e.g., the sequence of actions ["UP", "LEFT", "LEFT", "UP"] is converted into ["U","L","L","U"]). After this, the *Move* plan continues as usual. If, for some reason, the solver can not find a plan, we immediately fail the intention.

To ensure compatibility with the multi agent mode, which is not specifically designed for the automated planning, we decided to use the planning on a probability base. In particular, for the single agent the default probability of computing a path using the automated planning is 50%, but it can be adjusted as needed when launching the agent's script. While, for the multi agent the planning probability is automatically set to 0% and can not be changed.

### 2.6.4 Planning Considerations

Unfortunately, the *Deliveroo.js* setting is highly competitive and rapidly evolving, thus making the automated planning impractical in this scenario. The most important problem is the time needed to compute a plan: even in the smallest maps we tested, the number of atoms is very high, and this generates a huge search space, which is highly problematic when using a generic planner without the possibility to introduce an adequate heuristic to guide the search process. For this reason, we added a dedicated flag that blocks the option generation phase when a plan generation is undergoing.

Another problem regards our main strategy: instead of revising the currently executed intention to adapt it to the changed context, we prefer to generate a new intention that replace the current one. This is problematic since, when the plan is ready, a lot of time has passed, therefore the environment is highly different from before, and, consequently, the option generation will probably push a new intention, which is the best one at the current time. For this reason, the agent will spend most of the time generating new plans that will never be executed. To solve this problem, we introduced a dedicated parameter that can be set when calling the agent's script to extend the effect of the previous flag to the entire *Move* plan. This enforce the agent to complete his *go_to* action before pushing another intention.

Despite our best attempts to make the planner work in this setting, we still believe that, by its nature, the generic automated planning is not suited for this situation. Despite testing multiple generic planners, even on a local machine, the time required to generate the plan is too high, therefore, the generated plans are, in most of the cases, no longer valid due to the different state of the environment. Still, we maintain the current automated planning infrastructure as proof of concept that planning is possible in an autonomous agent context and believe that an adequate automated planning structure can be a powerful tool in different situations.

# 3 Multi Agent

This second part regards the development of the multi agent, in particular, the addition of a second agent (*"pal"*) that is able to operate in the *Deliveroo.js* setting. The two agents should be able to communicate, update the their internal beliefs with the shared information, coordinate and negotiate common solutions (e.g., decide who should pick up a certain parcel or collaborate to achieve goals).

Moreover, as said previously, the multi agent script is the same as the single agent one, and the multi agent mode can be enabled with a parameter when

launching the agent's script.

## 3.1 Additions to the Single Agent

With respect to the standard single agent components, several important components were added to ensure the multi agent compatibility. Doing so, we allow the multi agent to be able to operate even in the absence of the pal, making it resilient to those situations in which the pal may unexpectedly disconnect.

Similarly to the single agent, also in the multi agent case we have two specific multi agent tokens with the associated ids, this allows both the agents (i.e., *agent1* and *agent2*) to communicate. Follows a description of the main additions with respect to the single agent.

### 3.1.1 Pal Memory

The first step is to define an additional memory to contain all the information about the pal, in particular, his id, his position, his current intention and the timestamp of the last update we received from him. The pal's id will be set at the beginning of the agent script, since both of the ids and tokens are hardcoded in it, while the other information will be set and updated via message communication.

Another component modified in this regards is the *reviseMemory* function. In particular, if the pal fails to communicate with us for a certain time window, we consider hil as disconnected and, therefore, we clear his memory information. This allows the agent, despite him continuing to send non-blocking "*say*" messages to the non-existing pal, to operating as a single agent assuming that the pal is nowhere and carries no parcels.

### 3.1.2 Communication

The most important difference with respect to the single agent is the presence of a communication part that allows the agents to send an receive messages. In particular, we have defined four types of messages, one of which will be described in details in a next section (3.1.4). The other three message types allow the agents to synchronize their beliefs and operate accordingly. Following our rapid-oping-generation strategy and after multiple failed attempts with structured message share, we decided to adopt a "*UDP-like*" message share strategy for these three message types: if the pal is listening, he will update his belief accordingly, otherwise nothing problematic will happen. Similarly for the message reception strategy: if the pal is operative and sends us updates, we will update our belief, otherwise we will continue to operate with our current belief without problems.

Here is a rapid description of the three "*UDP-like*" message types:

- ***MSG_positionUpdate***: in every *onYou* callback, we use this kind of message to share the agent's current position with the pal. When the agent receives this message type, he will update his belief with the pal's new position, will set the new *last update* from the pal, will update the pal's entry in the *agent memory* and use the new pal's position to update its internal *time map*, allowing him to explore different parts of the map with respect to the pal;

- ***MSG_memoryShare***: after every *memory revision*, the agent sends the revised verion of his memory to the pal, in particular the *parcel and agent memories*, the parcels he is currently carrying and, again, its updated position. On the other hand, when the agent receives this type of message he will save the pal's carried parcels in a dedicated variable, he will remove from his *parcel memory* the parcels that, in his belief, are carried by the pal (if the pal is still carrying them, the agent will add these entries back to his parcel memory), and then, update his *parcel* and *agent* memories based on the information shared by the pal. Finally, he will call the handler of the *MSG_positionUpdate* type to update the pal position;

- ***MSG_currentIntention***: when the agent pushes a new *next intention*, he will also send it to the pal. If the agent receives this message, he will update the current pal's intention.

### 3.1.3 Options and Filtering

The presence of a collaborative pal, allows us to introduce some improvements during the option generation and filtering procedures. The first regards the best pick up option generation, in particular the reward computation. When the agent process all the single parcels, he will compute the reward of the single parcel, both for him and the pal, using his internal information about the pal without any communication. If the pal, for some reason, is not present in the agent's memory, his reward will automatically be zero. This additional pal reward allows the agent to perform additional filtering on the parcel: if the agent's reward is bigger than the pal's reward or the rewards are the same but his distance to the parcel is lower, then he will generate an associated option. Furthermore, if the current pal's intention is a "*go_deliver*", the agent know that the pal will probably ignore all the parcels and, therefore, he should always generate an option.

The second, and most important, improvement regards the introduction of a new option type: "*share_parcels*". This particular option is designed to solve "*corridor-like*" situations, where one agent can pick up parcels but not deliver them, while the other agent can deliver parcels but not pick them up. This option is pushed if no *best option* is identified before (there are not "*go_pick_up*" or "*go_deliver*"), if the agent is carrying parcels and there is a path to reach the pal. Furthermore, to handle the situation in which the path to a delivery is obstructed only temporarily
TODO: RICONTROLLA CODICE PER CASO SU TELEGRAM E FINIRE FRASE SOPRA
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)

### 3.1.4 Plans

spiegare piani aggiunti per scambio carried parcels spiegare i vari messaggi e il tipo di comunicazione che usiamo