Autonomous Software Agents Project Report

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

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

August 25, 2025

1 Introduction

This document contains the final report for the Autonomous Software Agents project, in which one or more agents are required to play the Deliveroo.js game. Agents developed for this project must implement the BDI (*Beliefs, Desires, Intentions*) architecture (Fig. 1), where each agent senses the environment to form internal beliefs, generates a set of possible intentions, and commits to one or more of them via a plan-based system, while continuously revising both intentions and beliefs.

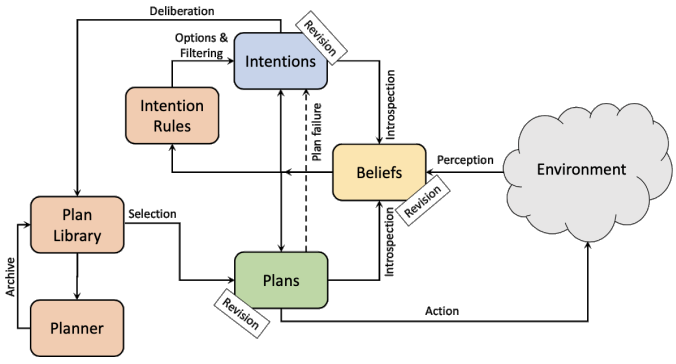

Figure 1: BDI architecture diagram used during the agent development

The project is composed of a single-agent part and a multi-agent part. The first part involves a single-agent implementing the basic functions necessary for correct operation. Specifically, the agent must represent and manage beliefs derived from sensing data, activate intentions, act on the environment, and use predefined plans to achieve its intentions. This occurs while continuously revising beliefs and intentions, allowing the agent to stop, hold, or invalidate the currently running intention. This enables the agent to respond appropriately in a rapidly evolving environment. Once these functions are implemented, the agent should interact with an automated planning utility to obtain the sequence of actions to perform.

The second part introduces a second agent capable of cooperating with the first one, and vice versa, to achieve the goal. Specifically, the two agents must be able to communicate, exchange beliefs, coordinate, and negotiate possible solutions to reach the goal, which may be unreachable by a single-agent.

Furthermore, both agents must operate effectively in different scenarios involving other competitive agents and rapidly evolving environments.

2 Single Agent

This section regards the development of a simple single-agent capable of performing the basic functions required to operate in the *Deliveroo.js* environment. Specifically, the agent must represent and manage an internal belief system constructed from sensory data received from the server, with the capacity to revise outdated or no longer valid beliefs. Based on these beliefs, the agent should be able to define and activate intentions, revise older intentions, and act in the environment to achieve them.

To achieve its intentions, the agent should use a set of predefined plans or integrate with a planner utility to execute the appropriate actions.

In particular, the single-agent operates using the same script as the multi-agent configuration, which is designed to be resilient to the absence of a second agent. In the single-agent setup, the agent avoids sending messages to the non-existent companion, and, in situations requiring information about the companion, such as its

position or its carried parcels, the agent treats the companion as null, assuming no location and no parcels.

2.1 Initial Connection

The first step is to initialize the agent's *belief set*, this includes the initialization of an empty memory. Then, the script evaluates the command-line parameters and initializes the connection using a specific single-agent token or, if necessary, a token generated on the fly. The script initializes the intention system, adds the predefined plans to its internal library and sets up the callbacks for *updates* from the server, such as "*onParcelsSensing*" and "*onYou*". Lastly, the script defines a *Promise* on the "*onMap*" and "*onConfig*" callbacks, awaiting their resolution before continuing the normal execution, specifically, with a temporized loop on both the "*agent's memory revision*" and "*option generation*" phases. The temporization for both loops is currently set at 50 ms: from our tests, this is the shortest possible time interval in which significant events may happen, with the lower bound corresponding to the agent's movement duration. This choice is justified by the rapidly evolving nature of the *Deliveroo.js* environment, keeping the overall approach simple and avoiding manual invocation of these functions.

2.2 Belief

This section covers the "*BeliefSet*" class regarding the single-agent. It consists of an initial setup phase, where the game map and the game configuration are defined, an "*environment sensing*" component where the agent's beliefs are updated using the information sensed from the environment, and a final "*memory revision*" phase, where outdated or unreliable information is removed from memory.

2.2.1 Initial Configuration

The first step includes the initialization of the internal memory. The key components are the "*agent memory*", which stores information about other agents, the "*parcel memory*", which stores information about parcels, and the "*me memory*", which stores information about the agent itself, such as its position, id, and token.

The second step defines an internal representation of the game map. Firstly, a matrix is created to store the tile type given their $[x, y]$ position. Then, each cell is validated, automatically marking unreachable cells, those lacking neighbors and thus unreachable, as "wall" (type 0). Lastly, using this matrix, a graph structure on the map itself is built, where each node represents a tile with its position, type, and neighbors. This structure enables efficient navigation using BFS. Additionally, associated to the game map, a "*time map*" matrix is defined, matching the its size, which contains the "last visited timestamp" for each tile. This is used during the exploration phase.

Finally, the game configuration is stored in a dedicated map for easier access during the game execution.

2.2.2 Environment Sensing

During the normal execution, the agent receives information from the server about sensed parcels, sensed agents and its own updated information. Upon receiving these messages, the belief set updates its internal representation to remain consistent with the current environment.

In particular, the "*onYou*" handler updates the agent's id (necessary when using a randomly generated token), its current position, and the time map. Accordingly to the current agent's position, all cells in the time map within the agent's field of view are updated with the current timestamp.

Regarding the "*onParcelsSensing*", the server provides the information on parcels sensed by the agent. This information is inserted into the internal parcel memory, updating existing entries for previously seen parcels and creating new entries for newly detected ones. A timestamp with the sensing time is stored with each entry to allow an accurate revision later. The same process is performed also for the "*onAgentsSensing*".

2.2.3 Memory Revision

The agent's belief may contain outdated information, such as a parcel no longer visible due to being outside the sensing range. While this information may remain relevant for a short period, it becomes unreliable over time. For this reason, the final fundamental component of the "*BeliefSet*" class is the *memory revision* system, whose purpose is to remove these old data.

In the *Deliveroo.js* setting, the server sends sensing updates only when events occur on existing and visible entities, such as a parcel appearing or an agent entering the sensing range, but does not notify the agent when entities disappear or move out of range. Consequently, if a parcel expires or an agent leaves the sensing area, no update is sent. To handle this, a *time window based approach* is adopted.

For parcel revision, all parcels in memory are cycled. If a parcel's position lies within the current sensing range

and its last observed timestamp is recent (within the *parcel's decay interval*, which is the minimal interval in which state changes about parcels may occur), the parcel is retained, as its presence is still reliable. If the timestamp is not recent, the parcel is assumed to have either expired or been collected by another agent, and so, is removed. If the parcel lies outside the sensing range, a larger time window is used before removal, since we cannot be sure if the parcel is still there or not. The same logic is applied to the *agent memory*, using the *agent's movement duration* as time interval.

2.3 Options and Filtering

To act in the environment, the agent must first identify possible and desirable options based on its current belief state. Once all options are identified, a filtering step selects the best one.

This process is implemented in the *optionsGeneration* function. It first identifies the best option (i.e., the one with the highest reward, see 2.3.1 for reward definition) by comparing the best "*go_pick_up*" option and the "*go_deliver*" option to the nearest reachable delivery cell, based on the agent's current beliefs. The selected best option is then compared to the currently executed intention, and, if it yields a higher reward, the current intention is stopped and replaced with the new option.

The function includes also control logic to prioritize the completion of "*go_pick_up*" options in some specific conditions. In particular, two conditions are identified: the first one is when no best option exists, to prevent pushing an "*explore*" option, the current intention is let to finish. While, the second one is when the best option is a "*go_deliver*", to prevent disruptive behavior such as switching direction to go deliver just before reaching a parcel to pick up, the current intention is let to finish.

An additional case is handled when there is no best option and the current intention is not "*go_pick_up*". In this situation, an "*explore*" option is generated using either a "*time-based*" or a "*distance-based*" strategy, with preference given to the "*time-based*" one.

Finally, the *optionsGeneration* function includes a flag to prevent executing it multiple times in parallel and performs a check to avoid running it during plan generation by the 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 any agent (i.e., a *free parcel*), a "*go_pick_up*" option is generated along with the associated estimated reward. The estimation accounts for both the reward of the parcel to pick up and the expected reward from the parcels currently carried. To compute this estimation, the total path, computed via BFS, is also considered. This includes both the route to pick up the parcel and the route to deliver it to the nearest reachable delivery. The agent's movement duration and the parcel decay interval are used to project the parcel rewards over time, and the last visit time for the target parcel is used to approximate its current score if it lies outside the sensing range. The resulting reward is scaled using fixed factors to prioritize proximity, based on the fact that picking up a nearby low-reward parcel is more cost-effective than traveling to collect a distant high-reward one.

Once all options are generated, the one with the highest reward is selected. In cases where multiple options have equal reward, the one closest to the agent is chosen.

If no best option is selected, typically because all options have zero reward, often due to all delivery cells being occupied by other agents, an alternative selection is performed: a simplified expected reward estimate is computed using the ratio between a 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 cell, the first step is to verify whether the agent is carrying parcels: if not, no "*go_deliver*" option is generated (no need to deliver). Otherwise, the path to the nearest delivery cell is computed. This path is then used to estimate the expected reward of the carried parcels, following the same logic used for the best "*go_pick_up*" option (see 2.3.1). If no path exists, it means that all delivery cells are blocked by other agents, and, therefore, the "*go_deliver*" option is not generated.

To prevent unwanted behavior, such as excessive preference for "*go_pick_up*" options, the number of moves performed by the agent while carrying parcels is tracked. This move counter is used, along with a fixed scaling constant, to increase the reward of the "*go_deliver*" option proportionally. This raises the likelihood of performing a delivery as the number of moves grows. The counter is reset after the delivery is completed.

2.4 Intentions

The intention management system is based on the code presented in the laboratories, with some key modifications. The most significant change regards "*intention*

revision": the intention queue has been removed in favor of maintaining only a current and a next intention. This decision reflects the fast-changing nature of the *Deliveroo.js* environment, where a queue of sequential intentions is likely to become obsolete before execution. The simplified model focuses on the current intention being executed and a next intention that is considered better and will replace the current one as soon as possible.

The async *loop* function in the "*IntentionRevisionReplace*" class serves the current intention, if existing. Once the intention is achieved, it is removed and replaced by the next intention, if any. The async *push* function receives a new option from *optionsGeneration* and determines whether to store it as the next intention. If so, the *stop* function is invoked on the current intention, causing its termination and enabling the *loop* to replace it with the new one.

Since *optionsGeneration* may rapidly push identical options multiple times, a mechanism is required to distinguish between new intentions (that must be replaced) and duplicates (that must be discarded). To address this, a comparison function, *areIntentionsEqual*, is used to compare the parameters of the new option with the predicate of the next intention, if existing, or the predicate of the current intention, if no next intention exists, discarding redundant entries.

2.5 Plans

Due to the fact that the code is split across multiple files, it was necessary to make the plans, defined in *agent.js*, accessible across modules. To achieve this, the plan library was created directly within the "*IntentionRevision*" class, where the plans are added. When the *achieve* function is called on an intention, the plan library is passed as a parameter to allow selection of the appropriate plan for execution.

The basic "*Plan*" class is the same as the one presented in the laboratories, with one modification: the plan library is also passed when invoking a sub-intention.

Using the "*Plan*" class, the predefined set of plans used by the agent to execute its intentions is defined.

2.5.1 Explore Plan

The "*Explore*" plan serves the "*explore*" intention. It consists of a "*go_to*" sub-intention aimed at exploring the map to discover parcels to pickup. The target coordinates are determined using one of two randomly selected strategies: time-based or distance-based. Both strategies first compute a list of explorable spawn cells using BFS.

The time-based strategy uses the *time map* from the "*BeliefSet*" to prioritize cells that have not been visited for a long time. The distance-based strategy assigns higher selection probability to cells distant from the agent. If the time-based strategy yields no valid target, the system uses the distance-based strategy. If no spawn cells are available, the destination is set as the nearest delivery cell, if available.

The probability of selecting cells in the time-based exploration strategy is computed as follows. First, the *age* of each explorable cell is calculated as the time in milliseconds since the cell was last seen, then normalized to the range $[0, 1]$. Concurrently, the *Manhattan distance* from the agent to each cell is computed, and a logarithmic transformation $\log(\text{distance} + 1)$ is applied to reduce scale disparity between age and distance. A softmax-like *priority* is then assigned to 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 testing, the optimal parameter values were found to be $\alpha = 5$ and $\beta = 1$. A target cell is then selected at random, weighted by the computed priorities.

2.5.2 GoPickUp Plan

The "*GoPickUp*" plan serves the "*go_pick_up*" intention. It consists of a "*go_to*" sub-intention that moves the agent to the target cell containing the parcel, followed by a pickup action. The pickup is executed through a wrapper that prevents multiple emit actions from occurring in parallel, manually marks the parcel as carried by the agent, and performs the pickup operation.

2.5.3 GoDeliver Plan

The "*GoDeliver*" plan serves the "*go_deliver*" intention. Like the "*GoPickUp*" plan, it consists of a "*go_to*" sub-intention to move to the target delivery cell, followed by a putdown action. The putdown is executed through a wrapper that prevents parallel emit actions, resets the moves counter, manually removes the delivered parcels from the agent's memory and adds them to an ignore list to prevent consistency issues, and performs the putdown operation.

2.5.4 Move Plan

The "*Move*" plan serves the "*go_to*" intention. It begins by computing a path from the agent's current position to the target coordinates using either BFS or an

automated planner (see 2.6). If a valid path is found, the agent follows it step by step until reaching the destination, where the plan concludes successfully. If no path is found, the plan fails.

During the path-following phase, several checks ensure the intention remains valid and relevant. For "*go_pick_up*" intentions, the agent verifies whether a parcel is still present at the destination. If not, the intention fails due to lost relevance. The path itself is also monitored: if a cell in the path is occupied by another agent for more than three consecutive moves, the path is considered blocked and the intention fails. Additionally, if the adjacent cell the agent intends to move into is currently occupied, the intention fails immediately. A final fail-safe manages scenarios where the agent is unable to move at all, and thus it is stuck, this also results in failure.

In all such cases, the intention is failed regardless of the situation. This aligns with the adopted strategy of frequently generated options: rather than attempting to fix invalid intentions, they are discarded, allowing a new and valid intention to be almost immediately selected and executed.

2.6 Planning

In the domain of autonomous software agents, plans can be automatically generated through *automated planning*, typically using the *PDDL language* and general-purpose planners. These planners require two inputs: a *domain file*, which defines the entities, actions, and rules of the domain, and a *problem file*, which specifies a particular instance of the environment, including the specific goal to achieve.

2.6.1 Domain Definition

The domain is defined in a dedicated domain file. In particular, the "*go_to*" intention is modeled using *automated planning* to compute the associated path. For this reason, the domain includes only the components necessary for movement, excluding components such as parcels and their scores.

The map is represented as a set of *tile* entities connected using directional predicates: *right*, *left*, *up*, and *down* (e.g., the predicate "*right ?t1 ?t2*" indicates that tile *?t1* is to the right of *?t2*). The *free* predicate identifies tiles not occupied by any agent. The agent itself is represented using the *me* predicate, and its current position is indicated using the *at* predicate.

As for the actions, the only ones necessary are the movement ones, named *right*, *left*, *up*, and *down* (distinct from the predicates with the same names). Each action takes three parameters: the agent, the current tile, and the destination tile. Using *right* as an example, the preconditions are: the agent is *me*, is located at the current tile, the destination tile is to the right of the current tile, and the destination tile is free. If these preconditions are met, the effects are applied: the agent is now at the destination tile and no longer at the initial tile; the initial tile becomes free, and the destination tile becomes occupied.

2.6.2 Problem Definition

Unlike the domain, the problem must be defined dynamically, as the environment changes continuously. To handle this, the problem definition is embedded directly within the agent's belief using the dedicated planning classes presented in the laboratories.

A new instance of the "*Beliefset*" class (distinct from the custom "*BeliefSet*" class) is created during the agent's "*BeliefSet*" constructor. When the agent receives map information from the server, it builds its internal map representation and then uses this to build the corresponding planning structure. Each walkable tile is represented as "*x<x coord>y<y coord>*", instantiated as a *tile* entity, initially marked as *free*, and connected to its neighbors using directional predicates.

As the environment evolves and the agents senses it, the planning belief must be updated accordingly. After every *onYou* update, the agent's position is updated by invalidating the previous *at* predicate, marking the previous tile as *free*, removing the *free* predicate from the new tile, and declaring the new *at* predicate. This update is also performed after every movement during the *Move* plan to ensure consistent and current state information. Similarly, other agents' positions are updated in the planning belief by adjusting the *free* predicate in both the *onAgentsSensing* handler and the *reviseMemory* function.

At this stage, the planning belief set is fully updated and ready to generate a problem file, following the method presented in the laboratories. A dedicated function was implemented to perform this operation. It takes as input the coordinates of the destination cell for the "*go_to*" intention and sets the agent being *at* that position as the problem goal.

After generating the problem file, a post-processing step is applied to correct an issue observed during testing. Specifically, the "*Beliefset*" class used in the laboratories, when performing an undeclare, adds negative atoms instead of simply removing the corresponding

positive ones. This behavior violates the *closed world assumption* and leads to semantic inconsistencies. Additionally, it causes the FF planner to crash if a negative atom appears in the final position of the problem's initial state. To resolve this, a regular expression is used to remove all negative atoms from the generated problem file.

2.6.3 Applying the Automated Planner

As previously discussed, automated planning is applied to compute navigation paths within the *Move* plan. Once the problem string is defined, it is passed, along with the domain file, to the *onlineSolver* function presented in the laboratories. This function returns a plan consisting of a sequence of actions from the agent's current position to the desired destination.

The resulting plan is then parsed: each action is converted into the same format used by the BFS-based pathfinding algorithm (e.g., a sequence such as ["UP", "LEFT", "LEFT", "UP"] is converted to ["U", "L", "L", "U"]). After this conversion, the *Move* plan proceeds as usual. If the planner fails to return a valid plan, the intention is immediately failed.

To maintain compatibility with multi-agent mode, which is not specifically designed for automated planning, planning is applied on a probabilistic basis. For the single-agent scenario, the default probability of using automated planning is 50%, but this can be adjusted when launching the agent's script. In contrast, for the multi-agent scenario, the planning probability is automatically set to 0% and cannot be modified.

2.6.4 Planning Considerations

The *Deliveroo.js* environment is highly competitive and rapidly evolving, making automated planning impractical. The primary issue is the time required to compute a plan: even in the smallest tested maps, the number of atoms is large, producing a vast search space. This becomes unmanageable when using a generic planner without the possibility to introduce a suitable heuristic to guide the search. To mitigate this, a dedicated flag was added to block the option generation phase during plan computation.

Another issue concerns the main strategy: rather than adapting the current intention to the changed context, a new intention is generated to replace it. This is problematic because by the time the new plan is ready, the environment has significantly changed. As a result, the option generation process selects a new intention based on the updated state, thus the agent repeatedly generates plans that are never executed. To address this, a dedicated parameter ("*-w*"), which can be set during the agent's script execution, was introduced. If set, this parameter extends the effect of the previous flag to the entire *Move* plan, forcing the agent to complete the "*go_to*" intention before considering new ones.

Generic automated planning, by its nature, remains unsuitable for this setting. Multiple planners were tested, even on local machines, but plan generation time remains excessive. As a result, most plans are obsolete by the time they are produced due to environmental changes. The current planning infrastructure is retained only as a proof of concept, demonstrating that planning is possible in autonomous agent contexts and, with an adequate structure and domain-specific heuristics, automated planning can be effective in different situations.

3 Multi Agent

This second part concerns the development of the multi-agent system, specifically the addition of a second agent (*"pal"*) capable of operating in the *Deliveroo.js* environment. The two agents must be able to communicate, update their internal beliefs with shared information, coordinate and negotiate common solutions (e.g., deciding who should pick up a specific parcel or collaborating to achieve goals).

As previously stated, the multi-agent script is the same as the single-agent one, with the multi-agent mode enabled through a parameter when launching the agent's script.

Compared to the standard single-agent components, several key elements were added to ensure multi-agent compatibility. This design allows the multi-agent system to function even in the absence of the *pal*, providing resilience in cases where the second agent unexpectedly disconnects.

As in the single-agent case, the multi-agent implementation includes two specific multi-agent tokens with associated ids. This enables both agents (i.e., *agent1* and *agent2*) to communicate. The following describes the main additions relative to the single-agent implementation.

3.1 Pal Memory

The first step is to define an additional memory structure to store all information related to the *pal*, specifically: its id, position, current intention, and the timestamp of the last received update. The *pal*'s id is set at the beginning of the agent script, as both ids and tokens are hardcoded. The remaining information is set and updated through message-based communication.

Another modified component is the *reviseMemory* function. If the *pal* fails to communicate within a defined time window, it is considered disconnected, and its memory data is cleared. This enables the agent, despite continuing to send non-blocking *say* messages to the now-nonexistent *pal*, to operate as a single-agent under the assumption that the *pal* is absent and carrying no parcels.

3.2 Communication

The most significant difference compared to the single-agent implementation is the presence of a communication component enabling agents to send and receive messages. Four message types are defined, one of which is described in detail in Section 3.4. The remaining three message types allow agents to synchronize beliefs and operate accordingly. Following a rapid-option-generation strategy, and after multiple unsuccessful attempts with structured message sharing, a "*UDP-like*" message-sharing strategy was adopted for these three types: if the *pal* is listening, it updates its beliefs, otherwise, no disruption occurs. Similarly, for message reception: if the *pal* is active and sends updates, the agent updates its beliefs, otherwise, it continues operating with the current belief state without issue.

A brief description of the three "*UDP-like*" message types follows:

- "***MSG_positionUpdate***": in every *onYou* callback, this message shares the agent's current position with the *pal*. Upon receiving it, the agent updates its belief about the *pal*'s position, sets the new *last update* timestamp, updates the *pal*'s entry in the *agent memory*, and modifies its internal *time map*, enabling spatial differentiation in exploration relative to the *pal*;

- "***MSG_memoryShare***": after each *memory revision*, the agent sends the updated version of its memory, specifically the *parcel* and *agent* memories, the parcels it is currently carrying, and its position. Upon receiving this message, the agent stores the *pal*'s carried parcels in a dedicated variable, removes from its *parcel memory* any parcels believed to be carried by the *pal* (re-adding them later if still present), and updates its *parcel* and *agent* memories with the received data. Finally, it invokes the handler for the "*MSG_positionUpdate*" message to update the *pal*'s position.

- "***MSG_currentIntention***": when pushing a new *next intention*, the agent also sends the intention name to the *pal*. Upon receiving this message, the agent updates its belief regarding the *pal*'s current intention.

3.3 Options and Filtering

The presence of a collaborative *pal* enables improvements in the option generation and filtering procedures. The first enhancement concerns the generation of the best "*go_pick_up*" option, particularly in reward computation. When processing individual parcels, the agent computes a reward for each parcel for both itself and the *pal*, using internal information without requiring communication. If the *pal* is absent from the agent's memory, the *pal*'s reward defaults to zero. This additional reward assessment allows for refined filtering: if the agent's reward exceeds the *pal*'s reward, or the rewards are equal but the agent's distance to the parcel is shorter, the agent generates an associated option. Additionally, if the *pal*'s current intention is "*go_deliver*", the agent assumes that the *pal* will ignore available parcels and will always generate an option.

The second and most significant improvement is the introduction of a new option type: "*share_parcels*". This option addresses "*corridor-like*" scenarios, where one agent can pick up parcels but cannot deliver them, while the other can deliver parcels but not pick them up. This option is pushed when no *best option* is identified (i.e., no "*go_pick_up*" or "*go_deliver*" options), the agent is carrying parcels, and there is a reachable path to the *pal*. To distinguish between temporarily obstructed delivery paths and persistent blockages, a dedicated counter is introduced. This ensures that the agent commits to a "*share_parcels*" intention only when the path is really obstructed, and not when the *pal* momentarily traverses a single blocking cell.

3.4 Plans

As previously stated, to handle *corridor-like* situations, a new "*share_parcels*" intention was introduced. It is triggered by the agent carrying parcels but being unable

to deliver them due to the *pal* blocking the delivery path. To prevent the generation of new options from disrupting the cooperation phase, a dedicated flag was added. This flag is activated during the execution of the following plans and blocks the push of new intentions while the collaboration phase is ongoing.

3.4.1 ShareParcels Plan

To handle the new intention, a dedicated plan "*ShareParcels*" was introduced. The first step in this plan is to send a "*MSG_shareRequest*" message containing the agent's current position, using the *emitAsk* utility to await the *pal*'s response. The *pal*, or the server, may respond in one of the following ways:

- **"timeout" or null**: in the first case, the *pal* fails to respond within the expected time window, resulting in a "timeout" from the server. The second case occurs in the single-agent setting, where the *emitAsk* wrapper automatically returns *null*. In both cases, the intention fails immediately;
- **"false"**: the *pal* explicitly refuses the request, causing the intention to fail immediately;
- **"you_move"**: the agent is obstructing the *pal* (handling the "*share_parcels*" intention requires a path of at least four positions between the agents). The agent must move to create sufficient space. If no valid path is available, the intention fails. Otherwise, once the agent has moved and sufficient space exists, it sends another "*MSG_shareRequest*" message;
- **"true"**: the *pal* accepts the request and sends the coordinates required for the parcel exchange, in particular a *wait position* (where the *pal* will wait), an *exchange position* (where the agent will drop parcels), and an adjacent *support position* (where the agent will move after dropping the parcels and await pickup by the *pal*).

Upon receiving a *"true"* message, the agent proceeds with the intention. First, it moves to the *exchange position* and waits for the *pal* to arrive at the *wait position*. If the *pal* changes its intention before arrival, this implies a failure in the parcel recovery plan (e.g., due to collision or path obstruction), and the agent's intention also fails. Otherwise, the agent drops its parcels at the exchange position, moves to the *support position*, and waits for the *pal* to complete the pickup. The intention is then considered successfully completed.

3.4.2 Handling the MSG_shareRequest Message Type

When the agent receives a "*MSG_shareRequest*" message, it first updates the *pal*'s position in memory and computes a path. If the agent is carrying parcels, the path includes also the nearest delivery cell before reaching the *pal*, otherwise, it is a direct path to the *pal*. If no valid path exists, the agent responds with *"false"* and the intention is rejected. Otherwise, the path is analyzed to extract the three required positions: the midpoint becomes the *wait position*, the next cell (from the agent's perspective) becomes the *exchange position*, and the following cell becomes the *support position*. If these positions are valid, the agent pushes a "*recover_shared_parcels*" option to commit to the exchange and responds with *"true"*, including the computed positions.

Due to the structure of this mechanism, the path must be at least four positions long. If this condition is not met, the agent checks whether it can reach a delivery position and whether the resulting path allows for at least four free positions for the exchange. If so, it commits to the movement and then retries the process. Otherwise, it responds with *"you_move"*, indicating that the *pal* must create space, and includes the number of required free cells.

3.4.3 RecoverSharedParcels Plan

To handle the "*recover_shared_parcels*" intention, a new plan "*RecoverSharedParcels*" has been introduced. Unlike the previous case, the agent already knows the three exchange positions. If a delivery is required (as determined by the message handler), it is completed first. The agent then moves to the *wait position* and waits for the *pal* to reach the *exchange position*, drop the parcels, and move to the *support position*. As in the "*ShareParcels*" plan, if the *pal* changes its current intention, indicating a failed share intention, the agent's intention also fails. Otherwise, the agent proceeds to the *exchange position*, picks up the parcels, and successfully completes the intention.

4 Results and Conclusion

This section presents final observations on the agent, including results obtained from various challenge maps tested extensively under tournament conditions, and a

potential direction for future improvements.

4.1 Results

A series of tests was conducted on selected maps from both challenges, chosen to evaluate key components of the single-agent and multi-agent systems. Multiple agent instances were executed in parallel to simulate competitive real-world conditions.

For the single-agent tests, four maps from the first challenge were evaluated using two modes: no planning (NP) and planning with 50% probability (5P). In the planning tests, the *-w* parameter was enabled to block new option generation during action execution. Prior tests conducted without this parameter, allowing the *option generation* process to run and potentially override the current action, generally yielded slightly worse results.

As shown in Table 1, cumulative score comparisons across the same map with and without planning indicate that planning degraded performance. The effect was most pronounced on smaller maps with narrow corridors or limited spawning tiles.

For the multi-agent tests, four maps from the second challenge and the hallway map were evaluated. Results are reported in Table 2. No major anomalies were observed: the agents operated effectively and exchanged parcels as required.

4.2 Possible Improvements

The agent does not perform sensing autonomously, instead, updated information is automatically sent by the server. This behavior is sufficient in most cases, but updates occur only when changes are detected. Consequently, internal predictions are required to maintain accurate beliefs. One example involves parcel expiration: the server does not notify agents when a parcel expires, requiring internal inference. Another involves other moving agents: when an agent observes another in motion, the server sends updates promptly. However, if the observing agent moves and loses visual contact, no further updates are sent. The agent must then infer that the other has likely moved and remove it from memory.

Both systems use a time-window approach (Section 2.2.3). This method is effective for parcels, however, for agents, a specific issue arises: if another agent remains stationary, the server sends a single update upon entering the sensing range. Without further updates, the time window expires and the agent is incorrectly removed from memory. While not critical, since agents typically remain in motion, this limitation suggests a possible area for improvement.

Map	Teams	First	Second	Third	Total Score	Percentage First
25c1_1 (NP)	10	230	210	190	1600	14.4%
25c1_1 (5P)	10	250	160	140	1270	19.7%
25c1_2 (NP)	10	1134	1094	1047	8164	13.9%
25c1_2 (5P)	10	534	483	455	3452	15.5%
25c1_6 (NP)	5	363	196	167	1005	36.1%
25c1_6 (5P)	5	189	179	175	739	25.6%
25c1_8 (NP)	5	833	694	680	3539	23.5%
25c1_8 (5P)	5	258	214	161	901	28.6%

Table 1: Testing results for the single-agent. In order, are reported the map name (presence of planning), the number of competing teams, first place score, second place score, third place score, total cumulative score of all teams and percentage of the first place score with respect to the total score

Map	Teams	First	Second	Third	Total Score	Percentage First
25c2_3	5	2172	1811	1721	8462	25.7%
25c2_5	3	1468	1432	1370	4270	34.4%
25c2_6	5	1941	1601	1579	7947	24.4%
25c2_7	5	1805	1738	1555	7859	23.0%
25c2_hallway	1	1471	-	-	1471	100%

Table 2: Testing results for the multi-agent. In order, are reported the map name, the number of competing teams, first place score, second place score, third place score, total cumulative score of all teams and percentage of the first place score with respect to the total score