ASA Project: Designing a BDI agent for Deliveroo.js

Team name: AutoMates

Richichi Andrea 257850
Salas Ichocan Rodrigo Alessandro 258774
Sorrentino Francesco 256151

A.A. 2024-2025

Table of Contents

Introduction

The aim of this project is to explore design and implementation of automated software agents in the context of Deliveroo.js educational game. The main challenge in this task is to create an agent that efficiently plans and executes tasks while being able to adapt to a quickly changing environment.

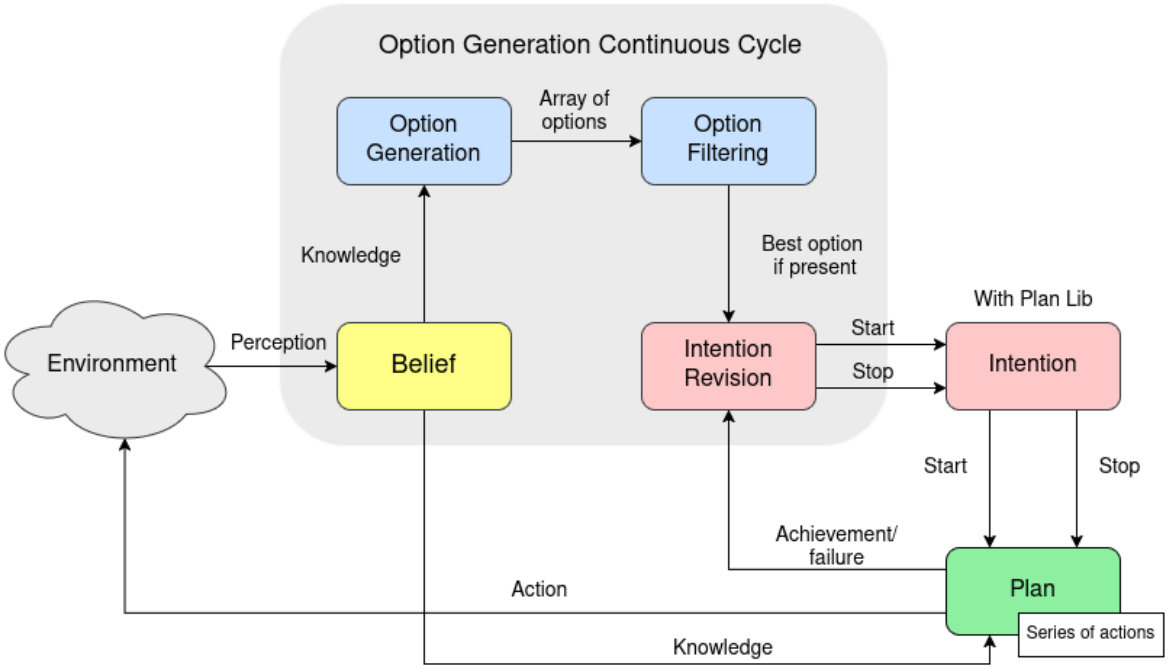
The project is structured in three main parts:

1. Single agent: creating a single agent using the BDI model as a blueprint. The different design choices will be discussed in every component of the agent.
2. Team of agents: after creating a fully functioning single agent, a team of two agents is created to cooperate in the game environment. This section will present both how the agents make choices and also how communication is handled.
3. PDDL extensions: The final phase of the implementation integrates PDDL to expand the planning capabilities of the agents. Here the extensions implemented will be presented by also formalizing the environment in PDDL.

Finally a testing and validation step of the software is performed. During the course the project has been tested in two separate challenges of which results will be reported, presenting useful insights obtained through interaction with other competitive agents. Further testing was performed after the challenges, making it possible to evaluate the project on a vast number of different scenarios.

Single agent

The base model for the agent is a modified version of the BDI model, which can be schematised as the image below. The B part is represented by the Belief component alone, the D part is represented by the rest of the option generation cycle and the I part is represented by Intention and Plan.



Two of the most important components of the system are the option generation cycle and Intention Revision, which are respectively the part where possible viable options are continuously generated and where intentions are managed.

Belief revision

In this section the way the agents revise its knowledge about the environment and generate possible courses of action is explored. Belief is managed through one main class, which thanks to callbacks can be revised as the Deliveroo game sends to it new data about the game. The following variables are the ones in the main Belief class, and they loosely resemble the callback functions.

```
config = {}; // Configuration data of the game
me = {}; // Info about the agent
time = {}; // Time inside the game
map = new DeliverooMap(); // Belief regarding the game map
agentBelief = new Map(); // Knowledge about agents' states
parcelBelief = new Map(); // Knowledge about parcels' states
```

In particular, a variable of type DeliverooMap is used to model the knowledge about the map of the game. The following are the variables used in DeliverooMap.

```
width; height;
map = []; // Map that gets updated with agents movement
originalMap = [];
deliveryTiles = []; // List of delivery tiles
spawnTiles = []; // List of spawn tiles
deliveryMap = []; // List of movement to reach the nearest delivery
spawnType = ""; // Type of map, based on spawn tiles disposition
mapBeliefSet = new Beliefset(); // Used for PDDL
POD = 0; // Parcel observation distance
```

Intention revision

This section is divided in two parts, which will explain how the option generation works and how it interacts with the intention revision. The option generation part works in parallel with the belief updates, and it operates within a continuous cycle, as illustrated in the picture. This setup allows the agents to continuously consider new courses of action and, as it will be explained later, change the current one.

The agent runs two main functions concurrently, *optionHandling*, which is called every certain amount of time which depends on if the agent works alone or in collaboration with another one, and *intentionRevision.loop*, which controls which intention the agent should pursue. The first stage of *optionHandling* is the generation of options.

Options are generated as objects that can have various structures, but the fundamental part is that all of them contain the *priority* and *type* variables, so that they can be compared with one another; the

rest of the variables are needed so that the planner can decide how to reach the objective. The generated options are of three types: PickUp, Delivery and Idle.

- Pickup(parcel) is used to calculate the priority of picking up a certain parcel. The priority is equal to *reward+alreadyCarried-decay-penalty*. *AlreadyCarried* is the total amount of already carried parcels, *decay* is the expected reward loss due to the time passing before delivering and *penalty* is an additional amount to encourage small deviation in order to collect more parcels and penalise more distant parcels.
- Delivery(x,y) is used to calculate the priority of delivering the parcels in a certain coordinate. The priority is simply the sum of carried parcels
- Idle is used to make the agent wander around the map if no other option is available. The priority is the lowest possible (-Infinity)

After these options are generated, there is a round of filtering, in which all options except the one with the highest priority are deleted. At the end the option is pushed into *IntentionRevision*.

In *intentionRevision.loop* there is the loop that extracts the highest priority option from a list, and checks if it is worthy to convert it into a plan to achieve. It does the following in order

1. If the option list is empty, don't do anything.
2. It checks if the first option in the list is still valid and it hasn't become obsolete. If it is obsolete, it deletes it. If the option is PickUp, it checks if the parcel still exists and that it hasn't been collected by someone else, while for Delivery it checks if the agent still carries parcels.
3. It tries to achieve the option, transforming it into a plan. When it stops for whatever reason, it deletes it.

Concurrently there is *optionHandling* that adds new elements into the option list. If the option is already present (e.g. Deliver to the same spot, or an Idle), it ignores it. If the addition of the option changes the ranking of those based on priority, the intention revision will stop the current intention.

Plans

When the intention revision accepts an option, the first thing that happens is the choice of the plan to follow. The *type* variable is used for that, which is a string that defines what kind of plan should be issued. The class *Intention* is used in the management of single plans, and is responsible for choosing the plan, executing it and, in case, stopping it.

```
#current_plan; // Contains the Plan instance
#stopped = false; // used to stop the Intention
#started = false; // Used for management of its lifecycle
#parent; // Used for management of its lifecycle
#predicate; // The option sent to the Plan to describe what to do
```

The plans can be divided in two categories, the ones used for multiple agents and the one for single agents

Single Agent	**Multi Agent**	**Both**
Idle, SoloDeliver, (PDDL)MoveTo, HeuristicsMoveTo, RandomMove	MultiMoveTo, MultiIdle, PDDLAlleyway, MultiDeliver	PickUp

An important thing to note is, in the case of single agents, it is possible to choose whether to use PDDL to plan the movement or not. Since the plans are made aware of the belief set, they are capable of understanding if during the course of their execution the objective becomes impossible to achieve. The plans, if needed, they can use sub intentions. The plans can be broadly subdivided based on the action.

- PickUp is one of the easiest, it calls a sub intention MoveTo, and if it is achieved, it tries to pick up parcels
- Idle decides which spawn tile the agent should go next, in order to explore the map. The tile is chosen by comparing the probability of new parcels spawning there (using spawn tile density) and the last time a certain parcel was last seen. It also calls a MoveTo sub intention.
- Deliver uses a special mechanism, in which it follows one between possible predetermined paths towards a delivery tile, calculated at the start of the game and if the path is blocked it waits. This system is used since delivery tiles, if occupied, are expected to be liberated soon. If the path is continuously blocked, it reverts to a MoveTo subintention.
- MoveTo tries to follow a path determined at every step, so that it can see sudden changes of the path. It uses the Astar algorithm to find a path, but in the case of a very trafficked map, this approach fails as it finds no direct path. For this reason there is a series of fallback plans in a hierarchy that are followed and can't be normally called, if not in these cases. A heuristic move is chosen if the usual one fails too often to move, in which it tries to go ahead regardless of the absence of the direct path, to at least getting closer to the objective. If even this approach fails, it tries to move in an almost random movable direction, trying to at least move out of a very trafficked place of the map.
- Alleyway will be described on the next chapter

The major difference between single agents' plans and multi agent ones is that there is a round of communication between the agents to decide a path which won't make them collide between each other.

Team of agents

As the complexity of the tasks and the environment increases, a single agent might not be sufficient in a real application. For this reason the next step of the project consists of designing a team of two agents.

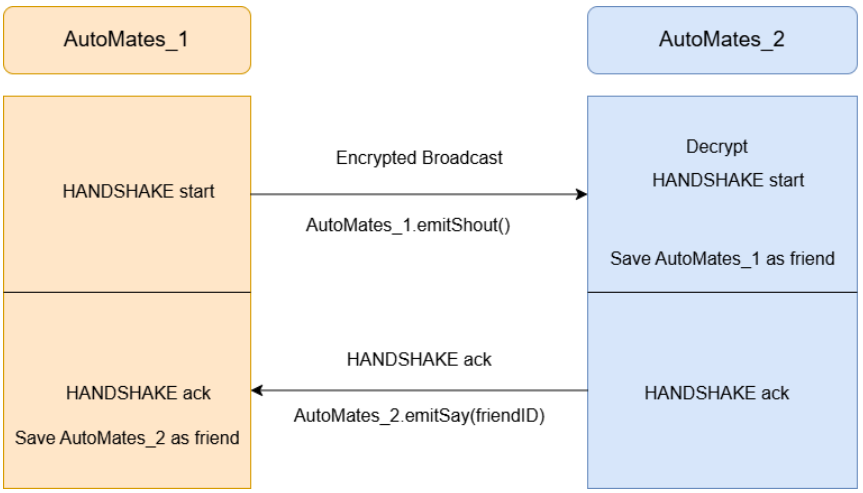
To collaborate properly the team needs both a communication protocol to be defined between the agents and also requires to properly handle cooperation when choosing which desire to fulfill given that both agents might have contrasting desires.

A particular scenario is relevant to properly test the design, the so-called "alleyway scenario", this section will present it but the complete resolution will be discussed in the PDDL extension part of the project.

Communication

Handshake

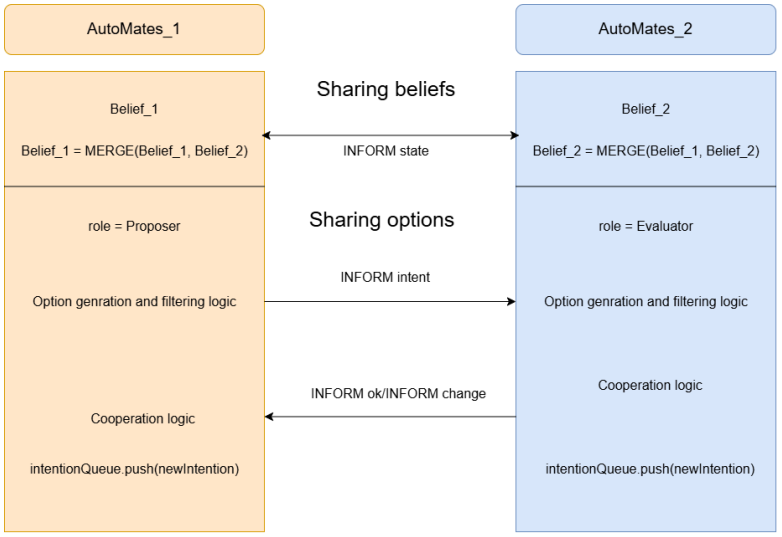
Before talking about how the agents share beliefs and perform option generation, a handshake protocol needs to be designed in order to properly let the agents connect to their teammate.



The diagram shows the basic protocol where one of the two agents that enters the game receives a broadcasted message and then responds with an acknowledgement.

Sharing beliefs and options

The agents in order to collaborate properly, need to also share their beliefs and options in order to decide the best actions to fulfill first given a certain environment state. Communication for this purpose is done as seen in the following diagram.



The idea behind this communication protocol is to have a more democratic approach in respect to a classic master-slave method.
The belief is shared between both agents since at regular intervals they send to their teammate a message where the belief set of agent and parcels is sent. Then a role for each agent is selected and the “Proposer” agent will ask the “Evaluator” agent if its intention causes conflicts or not and the evaluator will tell him, based on some cooperation logic, if it must be changed or not with a reply message.
After the agents have decided on an intention they push it to intention revision and try to achieve it.

Communication in alleyway scenario

The alleyway scenario has a similar protocol to the previous one. Based on the position in the alleyway the agents pick a role that is either “collector” or “deliverer”.

The collector agent creates a plan to successfully deliver the parcel and passes each action to the other agent with an “ALLEWAY act” message, then the deliverer fulfills the action and returns an ok message if the action was successful, letting the collector move to the next step in the plan. The communication ends when the plan is over.

Collaboration

To enforce collaboration between the agents we must define a clear strategy. This project aims to both maximize the score but also try to reduce at minimum any conflict between the agents that might cause for example bumping into the other or blocking them.
This is achieved by defining the two roles of “Proposer” and “Evaluator” and then make it so:

- There is no conflict between intentions of the agent
- The agents will take non-conflicting paths to achieve their intention.

The first point is achieved by having the proposer ask the evaluator if its intention is to be changed and then the evaluator responds by comparing its best option with the one the proposer sent. This comparison then decides based on the priority score, which agent must change intention in order not to cause a conflict.

The second point is much more complex since we must make sure one agent “books” a given path in order not to collide or get blocked. This is fulfilled by making the evaluator send to the proposer in the reply the path it will take and then making the proposer compute a path to avoid him.

PDDL Extensions

PDDL is very useful when dealing with complex situations that are hard to solve by coding them directly. To have a successful PDDL planning component a correct and clear formalization of the domain must be done.

Domain formalization

The game domain can be formalized by using the following predicates:

- tile: identifies a tile of any type inside the game
- agent: identifies an agent inside the game
- parcel:identifies a parcel
- at: tells where an object is (agent or parcel at a tile)
- right/left/up/down: used to decide tile locations
- occupied: used to set if a tile is occupied by an agent or not

Given these predicates we have also to formalize the four movement actions, pick up and put down.
A movement action is formalized as follows:

```
(:action right
    :parameters (?me ?from ?to)
```

```
    :precondition (and  (tile ?from)  (tile ?to) (agent ?me) (at ?me ?from)  (right ?from ?to) (not (occupied ?to)))
    :effect (and  (at ?me ?to)  (not (at ?me ?from)) (occupied ?to) (not (occupied ?from))))
```

This is because we can move only in tiles not occupied and after we move we update the occupied tile accordingly. This is the same for the other three directions.
Pick up and put down actions are formalized as follows:

```
(:action pickup
    :parameters (?me ?p ?t)
    :precondition (and  (tile ?t)  (agent ?me)  (at ?me ?t)  (at ?p ?t)  (parcel ?p))
    :effect (and  (at ?p ?me)  (not (at ?p ?t)) ))
  (:action putdown
    :parameters (?me ?p ?t)
    :precondition (and  (tile ?t) (agent ?me) (at ?me ?t) (at ?p ?me)  (parcel ?p))
    :effect (and (at ?p ?t) (not (at ?p ?me))))
```

Here we set a new position for the parcel depending on the action, maintaining the constraint from movement action that we cannot go in an occupied tile.
After presenting the domain, the problem scenarios must be formalized accordingly.

PDDL for pathfinding in single agent

The first simpler extension is to employ PDDL for computing a path in the single agent scenario. While this in practice is slower than traditional pathfinding via A* because of the use of an external planner, it is a much more readable solution and with very few changes, could be adapted in more complex domains (i.e. a map where obstacles could be moved).

PDDL for solving the "alleyway" scenario

A far more complex problem is that of the alleyway scenario. This scenario sees the team of agents inside an alleyway where on one side there is a parcel spawn tile and on the other a delivery tile. The choice to solve this using PDDL was made because in such a complex scenario is much more readable and shows off the power of PDDL.

The main difference to the previous scenario is that the position of the friend agent and the starting parcel position need also to be set. Also the goal is to deliver the parcel on the targetPos that is the delivery tile.

```
// Get objects of the problem
const objectList = [...myBelief.map.mapBeliefSet.objects, myAgentName, friendAgentName,parcelName];
const objects = objectList.join(' ');
 // Set positions on the map
 const initState = myBelief.map.mapBeliefSet.toPddlString()
+ ` (agent ${myAgentName})`
+ ` (at ${myAgentName} ${myCurrentTile})`
+ ` (agent ${friendAgentName})`
+ ` (at ${friendAgentName} ${friendCurrentTile})`
```

```
+ ` (parcel ${parcelName})`
+ ` (at ${parcelName} ${parcelPos})`;
// Construct target goal predicate
 const goal = `and (at ${parcelName} ${targetPos})`;
```

The resulting plan will have a sequence of action performed by either collector or deliverer.

Testing and validation

To validate the behavior and performance of our agents, we conducted both internal testing and external evaluation through two course-wide challenges. Internal testing focused on verifying the correctness of agent logic, including option generation, communication, and planning. The challenges, on the other hand, provided a competitive and dynamic environment where our agents were evaluated against others, revealing critical edge cases and performance bottlenecks.

The following sections summarize the main observations and issues encountered during the challenges, along with the modifications that followed.

Challenge 1 – Single-Agent Performance

Key issues observed:

- **Low Option Generation Frequency**: The agent was too slow in updating its available actions, reacting poorly to changes in parcel positions and game state.
- **Execution Latency**: Movement and decision execution were noticeably slower than other teams, reducing competitiveness.
- **Ineffective Parcel Prioritization**: The scoring function did not sufficiently penalize low-value or distant parcels, leading to inefficient choices.

Post-challenge improvements included increasing the option generation rate and refining the priority function with more accurate decay and distance penalties. These changes led to a more responsive and reward-oriented behavior.

Challenge 2 – Multi-Agent Coordination

Key limitations identified:

- **Invalid Plans in Complex Environments**: In narrow or high-traffic areas, agents failed to generate feasible plans due to incorrect assumptions about path availability.
- **Over-Conservative Collision Handling**: Our collision avoidance strategy was stricter than that of other teams, causing agents to cancel or delay actions unnecessarily.

In response, we improved the robustness of the option generation logic and replaced the blocking-based collision management with a more adaptive, negotiation-driven approach. Agents now communicate their intended paths and resolve conflicts proactively, which significantly improves coordination and flow in constrained maps.

Testing Results

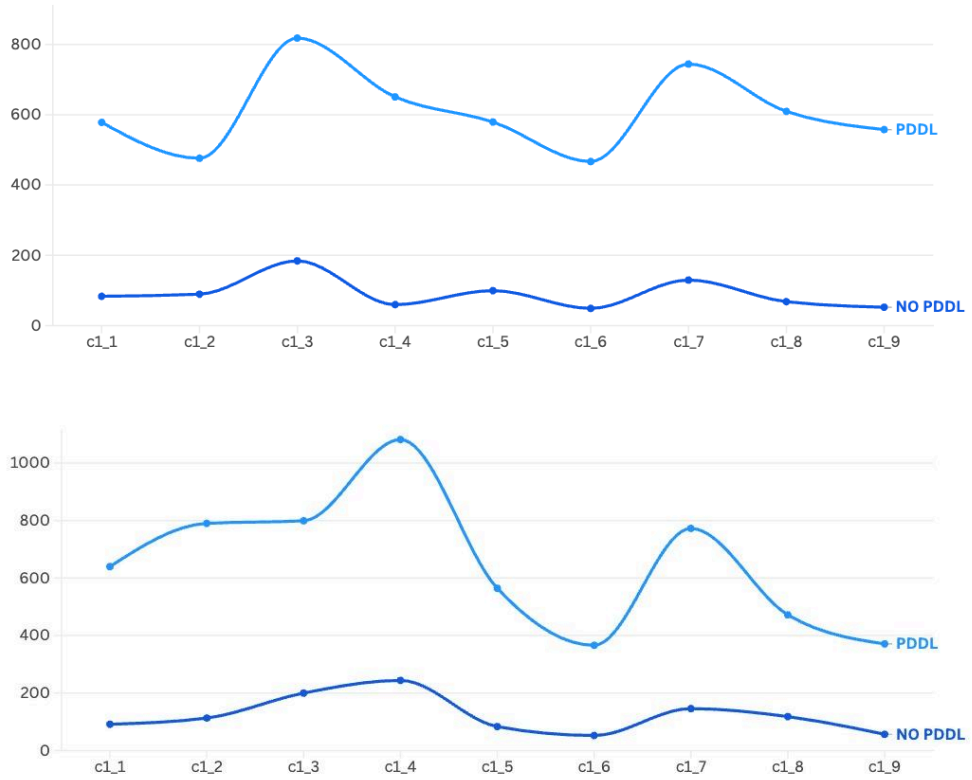
We conducted structured experiments on two map sets,one for single-agent and one for multi-agent agents,each consisting of 9 runs per academic year (24 and 25). The goal was to evaluate the computational performance and efficiency of our planning strategies.

Single-Agent Execution Time (Maps c1_1–c1_9)

Two variants of the agent were tested on maps from sets c1_1 to c1_9:

- **Standard (no PDDL)**: using built-in pathfinding and option generation.
- **PDDL-based**: using an external PDDL planner to compute navigation and task plans.

The charts below (Figures 1 and 2) display the **execution time per run**, not the agent score.



Note: The higher execution time for the PDDL agent is expected, as it relies on external API calls to compute plans.

Observations:

- **PDDL introduces significant computational overhead**, with average times several times higher than the standard version.

- The increase is more pronounced in maps with complex layouts (e.g., c1_3, c1_4, c1_6), where more planning is required.

- Despite longer runtimes, **PDDL offers more reliable and deterministic plan structures**, which is useful in controlled environments or for high-stakes decisions.

Multi-Agent Execution Time (Maps c2_1–c2_9)

Multi-agent tests were executed on map sets c2_1 to c2_9 for both years. These tests do not involve PDDL, coordination is handled internally through intention negotiation and communication.

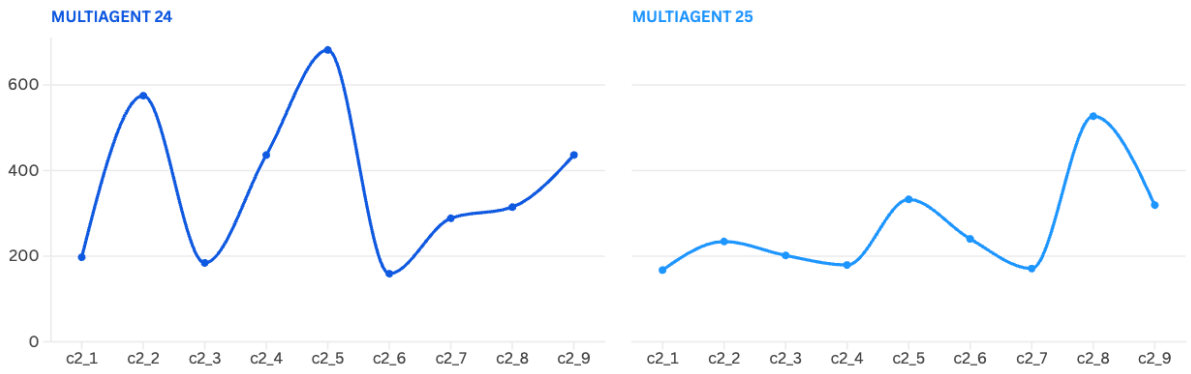
Figures 3 and **4** report the **average execution time per run**, capturing the cost of real-time negotiation, communication, and collision avoidance.

Observations:

- Execution times vary more widely than in single-agent mode, especially on Map Set 24.

- The variability reflects the **dynamic nature of agent interactions**, where blocking, rerouting, or resynchronization events introduce unpredictable delays.

- Map Set 25 shows **more consistent execution times**, likely due to simpler geometry or fewer bottlenecks



Conclusion

Designing this project has provided great insights into the world of BDI agents operating into dynamic environments. By implementing both a single and multi agent scenario we explored the complexities of belief, desire and intention revision as well as the issues in communication and cooperation. By extending the project with PDDL, more complex situations were solved and also the agent benefited from more clear formalization of the environment.