Autonomous Software Agent Project

Multi-agent Deliveroo system

Andreas Chini [258909 - andreas.chini@studenti.unitn.it]
Enea Strambini [248437 - enea.strambini@studenti.unitn.it]

July 2025

Abstract

The scope of this project was that of developing an Autonomous Software Agent, able to navigate an environment in continuous evolution. The Agent uses a Belief Desire and Intention (BDI) architecture, by analyzing the environment, developing intentions based on the entities sensed, and using a predefined plan to satisfy them. The project first started with a single agent, which by the end evolved to a dual agent setup, with cooperation between the two. The use of PDDL was also explored as an additional feature during the development of the project.

1 Introduction

In this report, we will present the structure and components of our *Autonomous Software Agent*, designed to compete in the *Deliveroo* environment. We will also explore some of the design choices and decisions that guided us through the development and refinement of the project. Finally, we will draw some conclusions and briefly consider what future steps might be taken to improve the *Agent*'s behavior. All the code, along with instructions on how to run it can be found in the *GitHub repository* [1].

The main goal of an agent competing in this challenge is to collect the highest amount of parcels in the shortest time possible. The environment setup can change slightly, mainly in the form of different maps. Each agent must be able to adjust its behavior to overcome these changes and also be able to navigate a map where other agents, adversary or otherwise, are also competing.

The environment where the agents must operate is designed to be challenging, composed of *tiles*, each with different characteristics. The tiles can either be walkable or not walkable, forcing the agent to consider this when planning to move across the map. Moreover, the walkable tiles can be further divided into three subgroups: *simple*, *spawn*, where parcels can appear, and *depots*, where an agent must deliver the parcels gathered.

The rough overview of the *Agent*'s execution loop can be divided into four steps. The first one is that of analyzing the **environment**, based on the information gathered; the *Agent* then defines a new **goal**, prioritizing parcel collection. After a goal is defined, the *Agent* creates a **plan**, used to achieve the original intention. Finally, the agent **executes** the plan, adapting to the changing conditions of the environment.

The development of this project started with a single *Agent*, with its basic behaviors and plans, able to obtain information from the environment and act accordingly. It then proceeded to include behaviors that allowed two agents to communicate with each other, allowing collaboration during the challenge. The use of PDDL was also explored, in particular to devise an alternative plan for the *Agent* to move to a specific position in the map.

2 Single agent

The first step in this project was the development of a single *Agent*, able to sense the environment surrounding it and make decisions accordingly. The single agent integrates a lot of measures to help it compete at its best.

2.1 The starting point

At the start of this project, there were two possibilities: either start completely from scratch or build our *Agent* on top of one of the basic templates. Due to scarce previous experiences with JavaScript, starting from a template was the best option, specifically the `Lab4 intention-revision.js`. Using something that already worked to then evolve into a fully working *Agent* allowed for an overall smoother experience than starting from scratch would have. The template used already contained a basic agent structure, parcels sensing algorithm, intention revision, and basic movement algorithm. All of these were a good starting point but were naturally outgrown during the development of the project, being replaced with our own version.

2.2 Macro tasks

The *Agent* is characterized by three of what can be called *macro tasks*: searching, gathering, and delivering the parcels. Each of these is then further divided into smaller tasks, with one or more plans used in order to achieve it.

Since the package decay is quite slow (*1 value/s*) in comparison to the movement speed (*between 2 and 10 tiles/s*), it was decided to always prioritize parcel pick-up, with respect to either movement or delivery. If the decaying speed were higher, this behavior would certainly be worse, forcing a reconsideration of the priority of the *Agent*; mainly delivering the parcels that it is already carrying, ignoring further parcels. Even though parcel pick-up has the highest priority, the *Agent* still considers the number of parcels it is carrying. If this exceeds a predefined threshold (`MAX_CARRIED_PARCELS`), the agent will prioritize the delivery of those parcels instead.

[1] GitHub repository: https://github.com/strambinienea/ASA-Automates

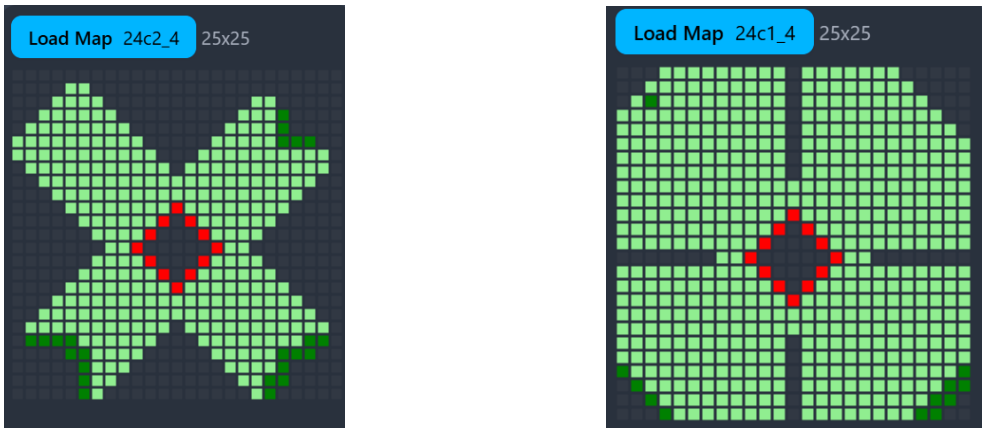

Figure 1: Maps where the random movement proved useful

The macro tasks, or *Intention*, are stored in the *Agent* in what's called the `IntentionQueue`. This is sorted to order the intention with the right priority, first parcel pick up, then parcel deposit and finally movement of the *Agent*.

If the *Agent* has no option in the queue to pick up or deliver a parcel, it will begin to search for it. It does that by using the `WorldMap` to get the spawn tiles and then selecting a random one between those. In particular, it will first select tiles that are near the *Agent*, under a certain radius, and in the case where none are near it, will then select a random one looking in the whole map. This is done in order to avoid moving on tiles where parcels cannot spawn, all the while being as near as possible to the spawning tiles.

This little trick was found very advantageous in maps like *Fig. 1*, allowing the *Agent* to always be near a lot of spawn tiles so as to have the highest possibility to pick up a parcel. This allowed us to get an edge on other agents, that continued to travel from a group of spawn tiles to another, losing chances to gather parcels.

A drawback of this particular feature appears on larger maps, filled with *spawn* tiles. The *Agent* slows down significantly in these situations because it has to check the distance, through the A* algorithm, to each tile. This issue can be overcome by balancing the radius the *Agent* will consider as a neighbor, through the environmental variable `MAX_DISTANCE_FOR_RANDOM_MOVE`. This becomes then a decision between the performance of the *Agent* and the performance of the code, and must be addressed map by map.

2.3 Studying the environment and BeliefeSet

A fundamental part for the *Agent* to work is that of studying the environment and updating its own *BeliefSet*. This is mainly done in the `world-map.js` and `world-state.js` files.

2.3.1 World Map

The `WorldMap` class is mainly responsible for storing information regarding the environment the *Agent* is competing in. This includes the map, with all the tiles' position and type, the parcels present in the map, and the position of other agents in the environment, be it adversary or friendly. These information are updated when some events are received from the *Deliveroo* client. This allows the map to be always up to date and offer the *Agent* the best knowledge of the environment.

Inside the world map, information is stored in separate data structures, mainly divided into three arrays: one for the whole map, one containing only depot tiles, and one containing only spawn tiles. This separation is useful because many functions only require knowledge of one type of tile, and it allows working with smaller data structures, further improving the performance of the operation.

Another method that sees a lot of use throughout the whole project is the `getWalkableTiles`. This method returns all tiles that are considered walkable around the map, effectively filtering out the wall tiles from the map data structure. It further filters the data, removing any other agent that is currently roaming the map and that has been seen by the *Agent*. This last filtering step can

be disabled by passing a boolean argument to the function; this comes in handy when the *Agent* needs to consider the map without any knowledge of other agents.

A limitation of the map is caused by the *Agent*'s observation distance, a value that limits how far an agent can sense entities in the environment. This means that sometimes the information with which the *Agent* is working is not correct but is an assumption made on past information. This could cause the agent to plan on incorrect information, but should be able to correct the course once new information are added to the map.

2.3.2 World State

The `WorldState` class is responsible for actually updating the `WorldMap` with information from the environment. It does this by waiting for events from the *Deliveroo* client, and uses callback functions to update the information in the `WorldMap`.

The information contained in these classes is enough for the *Agent* to make informed decisions, creating new intentions and planning accordingly to the entities that populate the environment.

2.4 Micro tasks

As touched on before, each macro task can be divided into one or more micro tasks, that define how the *Agent* will complete it. These are represented in the project by the `Plans` classes. A *plan* is a set of instructions that the *Agent* follows in order to complete an action; one or more plans are necessary in order to complete a macro task, also referred to as *Intention*.

Four plans are defined in the project:

- `GoPickUp` - This plan defines the actions the *Agent* has to take in order to pick up a parcel.
- `GoDropOff` - This plan defines the actions the *Agent* has to take in order to deliver a parcel.
- `GoTo` - This plan defines the actions the *Agent* has to take in order to reach a specific position in the map. This plan is used as a *sub-intention* in both `GoPickUp` and `GoDropOff` plans in order to move the *Agent* to the target location.
- `GoToPddl` - This plan is a variation of the `GoTo` plan, that use PDDL in order to find a viable path through the map, to the destination.

2.4.1 Path finding with A* Algorithm

In order to move effectively throughout the environment, the *A* Algorithm* was used to find the best path in the map. The function tasked with finding the best path only consider *free* tiles, these are tiles that can be walked upon by the *Agent*, and that are free from any other agent competing in the map. This is possible thanks to the `WorldMap` class, that exposes all these information for the *Agent* to use.

```
// Move the agent, try to move a couple of times, with a brief delay in the middle
let hasMoved :boolean = false;
let result;
for ( let attempt :number = 0; attempt < 2; attempt++ ) {

    result = await this._getClient().emitMove(direction);
    if ( result ) {

        Logger.debug("Move from ", agentX, ", ", agentY, " to ", x, ", ", y, " successful");
        hasMoved = true;
        break;
    }
    await new Promise( executor: res => setTimeout(res, timeout: 10));
}
```

Figure 2: Retry moving if path is blocked

One improvement that gave us an edge in the competition against other agents is the fact that the *Agent* will retry the move a couple of times if it fails (Fig. 2). This means that if the path is obstructed by another agent, the *Agent* will retry to move on the same path after a brief delay. Since most other agents implement a collision avoidance algorithm, they will most likely change path, allowing the *Agent* to follow the original path without the need to recalculate a second route. This is very advantageous in maps like Fig. 3 because the *Agent* is forced to pass through a lot of corridors both for pick-up and delivery.

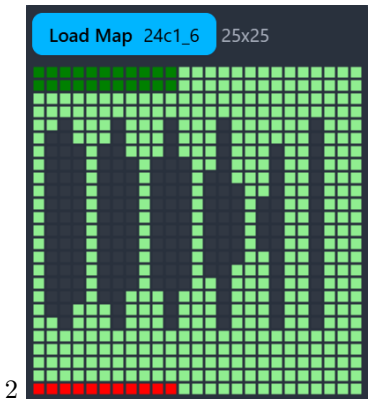

Figure 3: Map 1.6

3 Double agents

The double agent scenario consist of spawning two agents in the environment. The code for each *Agent* is exactly the same, as seen in the single agent section. Some routines and checks are in place that only activate on a double agent scenario, allowing the streamlining of some operations, and a degree of cooperation among the *Agents*.

Even though the agents are often referred to as *Leader* or *Follower*, the architecture organically moved away from a hierarchical structure, often ignoring the 'status' of the *Agent*, towards a first-come, first-served behavior, particularly when considering parcels competition. The hierarchy of the *Agents* is mainly used at the start up, where the *Leader* is responsible for deciding the *Agents* behavior, also called *Hand2Hand* mode, that will be explained later on the report.

3.1 Avoid competing for parcels

In order to improve the performance of each *Agent* in the competition, it is fundamental that they harness the ability to communicate with each other. This is mainly used to avoid internal competition when picking up parcels. Without this feature, if both *Agents* were in the vicinity of the same parcel, they would both compete to collect it first. This is avoided thanks to the ability of *Agents* to send messages to each other.

```
/** @type { ParcelsToIgnoreMessage } */
const message : ParcelsToIgnoreMessage = {
    action: "multi_pickup",
    parcelIds: parcels,
}
```

Figure 4: Structure of the ParcelsToIgnoreMessage

The first *Agent* that spots a parcel and develops the intention of picking it up will communicate this intention to the other *Agent* via the `ParcelsToIgnoreMessage`. This is done when sorting the intention queue and will let the other agent know what parcel to avoid picking up. These parcels will then be inserted into the `parcelsToIgnore` list, which is specific to each *Agent*.

Each *Agent* will then filter the parcels during the option generation phase, avoiding parcels present in the `parcelsToIgnore` list.

```
options.forEach(option => {

    if ( option[0] === 'go_pick_up' && agent.parcelsToIgnore.includes(option[3]) ) {
        Logger.debug("Ignoring option to pick up parcel: ", option[3], " as it is in the ignore list.");
    } else {
        agent.push(option);
    }
})
```

Figure 5: Filtering parcels at option generation

This is fundamental to decrease the number of moves and wrong intentions each *Agent* develops during its lifetime, and greatly increase the performance of each.

3.2 Sharing Agents position

Another improvement that can be achieved thanks to message exchange is that of better knowledge of the other *Agent* position. Both *Agents* will share their position with one another, allowing to fill in a gap in the `WorldMap` caused by the agent observation distance. The exchange of the *Agent* position is fundamental for the initialization of the *Agents*, since without this the *Leader* would work on a map with possibly incomplete information, in the form of not knowing the exact location of the other *Agent*.

3.3 Special case: corridor-like maps

A particular kind of map could render impossible the navigation in case of dual agent mode. If both *Agents* found themselves on the same *corridor*, they would be unable to deliver and gather any parcel. For these reason a special mode was developed, called *Hand2Hand* mode.

3.3.1 Entering Hand2Hand mode

At the startup of the program, the *Leader* will start the initialization phase. This phase will start once the *Leader* receives the *Follower* position, allowing to work on a complete map. Knowing both *Agents* position the *Leader* can safely assume to know any relevant information. It will then check if it is able to reach at least one *depot* or *spawn* tile. If it can't reach either one of those, then it means it's stuck in a corridor and must enter the *Hand2Hand* mode. This is done by changing a variable in the *Agent*, called `Hand2HandMode`, to be either `GATHER` if the *Agent* can reach a *spawn*, or `DELIVER` if it can reach a *depot*. The *Agent* that initialized the *Hand2Hand* mode is also responsible for notifying the other *Agent* of this change in behavior, also communicating to it the new role it will have to fulfill, meaning the remaining mode between `GATHER` and `DELIVER`.

3.3.2 Different option generations

Each of the two mode has a different option generation function, based on the expected behavior of the particular mode.

- `DELIVER` - The *Agent* in this behavior is able to reach a *depot* tile, but not a *spawn*. The first step it takes is that of moving to the depot tile, giving the other *Agent* the most amount of space to move in the map and gather parcels. It will then search for a common delivery point, a point that is reachable by both *Agents*, where the *Gather Agent* will hand off the parcels for the *Deliver Agent* to deposit in the depot. Once a common delivery point is found, this *Agent* will wait at the depot, scanning the delivery point for any new parcels, and picking them up once found. It will only consider picking up parcels in the common delivery point, discarding any other pick-up intention.
- `GATHER` - The *Agent* in this behavior is able to reach a *spawn* tile, but not a *depot*. This *Agent* will wait for the common delivery point to be chosen by the *Deliver Agent*, before starting its own routine. The reception of the delivery point will happen through a message called `DeliveryTileMessage`, where the *Gather Agent* will check if it can reach the delivery tile. Once the delivery point is agreed, the gather agent will collect all the parcels in the map, depositing them to the delivery point. If no parcels are available to pick up, the agent will move to one of the spawn positions and wait for new parcels to appear in the map.

3.3.3 Choosing the common delivery tile

The *Deliver Agent* is tasked with finding a common delivery tile, reachable by both *Agents*, where the hand-off of the parcels can happen. This is done through a function that moves through all the walkable tiles in the map, starting from the depot and crawling through all the neighboring tiles.

4 PDDL

The PDDL in this project was embedded in the most used plan, the `GoTo` plan, used to solve basically all of the intentions an agent can develop. PDDL perfectly adapts to the characteristics of this plan, that involve navigating the map, searching for a viable path between two tiles.

The first step for including the PDDL in the project is that of understanding the problem in order to develop an appropriate `domain.pddl` file, that takes into consideration the map entities, as well as the agent options in the map.

Once this is done, the *predicates* can be outlined; these answer questions about the status of the map and the entities that populate it. Among the outlined predicates, there are `(on_tile ?tile)`, that is used to infer if the agent is currently on the given tile, or `(above ?from_tile ?to_tile)`, used to check if a given tile is above another (this predicate exists for all four directions).

Once the predicates are defined, allowing the agent to understand the map, the *actions* must be outlined; these define how the agent will react to the map condition, and how to act in order to achieve its goal. Four actions are defined, one for each direction, in the form of

```
(:action mv_up
    :parameters (?from_tile ?to_tile)
    :precondition (
        and (on_tile ?from_tile)
        (not (on_tile ?to_tile))
        (below ?from_tile ?to_tile)
        )
    :effect (and (on_tile ?to_tile) (not (on_tile ?from_tile)))
)
```

This action requires two parameters, both tiles. It then checks if the conditions are valid in order to apply the action, namely that the *Agent* is on the `from_tile` and not on the `to_tile`, and that the first tile is below the second, in order for the agent to move up. If the conditions are satisfied, then the *effect* can take place, which requires the *Agent* to be on the `to_tile` and not on the `from_tile`, essentially moving up.

4.1 Problems with the PDDL implementation

A problem that has arisen when using the PDDL movement plan is that of *Agents* bumping into each other. This is due to the delay introduced by the PDDL client when solving the problem. The *BeliefSet* used to find a solution becomes outdated by the time the client returns a solution. This means that in some cases a tile that was free at the start of the computation is no longer free, or a parcel to pick up has expired by the time the *Agent* reaches it. Though an evident problem, this is not easily fixable unless one implements its own local client, but can be mitigated by proper agent behavior.

5 Conclusions

The development of an *Autonomous Agent* has not been an easy task. Learning a new programming language, deciding the best course of action, and trying to find and implement a unique solution to have the edge over the competition were only some of the challenges we met. Things became even more complex with the addition of a second agent and the communication systems. But the satisfaction of seeing everything work and, even more, winning the competition, was worth the hours of troubleshooting.

Nonetheless, the implementation is not perfect, the coordination required for both agents to work together means that sometimes the whole execution fails, due to an unexpected behavior, and makes it hard to replicate, since the error is caused by a small margin. The whole code in general could surely be trimmed down to a more manageable size, removing redundant functions or unused behavior, and overall could be significantly improved from a performance point of view. Since performance was not the main focus though, we considered a compromise, putting it into the back seat in favor of features.

Another aspect that could be improved is the PDDL, mainly the need for an external client used to solve the problem. This slows the execution down significantly with respect to the normal

performance of the project. This issue could be addressed by hosting our own PDDL solver; that would have sped up the execution significantly, but the complexity and time constraint didn't allow us to properly deploy this feature.

But even with this problems the *Agent* works quite well, and with all the little adjustments and features it allowed us to win the challenge. Our agent was not the best in every map; there were, in fact, some techniques implemented by our opponents (like backtracking a few tiles and then retrying, when a collision happens) that gave them the edge in one or two maps, but slowed it down in the others. So overall we got the best placement by being somewhat good at everything while not excelling in one particular thing. And that that school of thought is what should be followed when creating an agent, because adaptability and capability of automatic choices is what differentiate a hard coded algorithm from an *Autonomous Agent*.