Autonomous Software Agents Report

Murtas Cristian, 248025, cristian.murtas@studenti.unitn.it
Wang Marco, 249368, marco.wang@studenti.unitn.it

July 2024

Abstract

This report presents the development and evaluation of a Belief-Desire-Intention (BDI) agent designed to autonomously play the game *Deliveroo*. In this game the agent is tasked with picking up parcels from various locations and delivering them to designated delivery zones.
The BDI framework was chosen for its robust mechanism and its ability to handle dynamic and complex environments through the integration of beliefs (information about the world), desires (objectives to achieve) and intentions (plans to accomplish the objectives).
The development process involved the creation of an environment model, the formulation of a strategy for efficient parcel collection and delivery, the implementation of the BDI agent and the integration with a PDDL planner. Furthermore, a collaborative version of the agents has been developed. The performance of the agent was evaluated through a series of tests that measured its ability to successfully complete deliveries and adapt to changing environment conditions.

1 Introduction

The objective of the project was to develop an autonomous software able to play the Deliveroo game on our behalf. The goal of the game is to pick up the parcels and bring them to a delivery zone to gain points.
We structured our agent following the BDI[1] architecture. The BDI gives an approach to designing systems that operate in an autonomous way, making decisions by sensing and interacting with the environment. Also, the agent is able to continuously revise and replan its intention to choose the best option accordingly to the circumstances. A multi-agent version has been developed as well.

2 Belief Management

Beliefs represent the state of the agent and what it knows about the environment. As they constitute the memory of our agent, the goal is to keep them consistent through continuous updates and revisions.

2.1 Implementation

The beliefs are maintained in the *Beliefs.js* file, which contains all the information that our agent needs to generate an intention. Their update happens in *Agent.js*[2]. We categorized the belief set into four groups: the beliefs subject to changes, constant of the map, communication buffer and a set of hyper-parameters.

[1]Belief - Desire - Intention.
[2]It handles the entire agent loop.

The *Deliveroo Api* provides the sockets to collect the data about the surroundings, allowing us to revise the belief set.
In the next sections we will explain how we leveraged the available listeners.

2.1.1 Map

Thanks to the *onMap* socket we receive the structure of the map when the agent connects. This information is used to create a representation of the map for latter intention generation and planning. Here we also initialize the heat map to keep track of the most dense areas in terms of parcel spawning. In this phase we also store all the delivery zones.

2.1.2 Agent Information

The *onYou* socket is used to receive all the data related to our agent. Specifically, through this channel we receive the most up-to-date information about our current position, score, name and id.
The first time we receive the position of our agent, we remove the parcel spawner and delivery tiles that are unreachable.

2.1.3 Parcel Sensing

The *onParcelsSensing* socket allows us to collect the data about the parcels entering our range. In particular, we applied a filter to retrieve only the parcels that are not carried by any agent and the ones that are not blacklisted[3].
Subsequently, for each of them we store their position, their score, their id.
During this step, the heat map is also updated. Indeed, when a parcel is sensed we increase the probability of the tile where the parcel has spawned.

2.1.4 Other Agents

We have been able to retrieve the information about the surrounding agents through the use of the *onAgentsSensing* socket. For each sensed agent we store its position and the score. Moreover, we added the *unseen* parameter, which is the elapsed time from the last time we saw it. This property is useful to understand the crowdedness around our agent.

2.1.5 Configuration

With *onConfig* we obtain the initial configuration of the game, which is used by our agent to help with the decision making process.

3 Intention Deliberation

The purpose of the intention deliberation is to maximize the points delivered by the agent through an efficient decision-making process that usually involves balancing trade-offs. This part is encoded in the *Intention.js* file and it represents the brain of our agent. We divided the implementation of the intention into four types:

- Pick Up: is the intention to pick up a parcel;
- Put down: is the intention to deliver a parcel;

[3]A parcel is blacklisted when it is temporarily out of reach, see chapter 7.

- Target move: move towards a tile specified by us. This is needed to explore the map;
- Stand still: is used to force the agent to stop on a given tile and wait for an awake signal.

Each intention is chosen based on the current understanding of the environment by the agent. If the updated information suggests a better intention that would yield more points, the current intention can be replaced or stopped.

3.1 PutDown

The precondition to deliver is that the agent is carrying at least one parcel, but this alone is not enough. We also apply additional rules and the agent might decide to deliver if:

- it is carrying a certain number of packets[4];
- there are no other packets around[5];
- there are packets around but the reward is less than the loss.

The calculation of the third point start by computing the reward decay, which represent the decay of one parcel for each of step of the agent:

$$\text{rewardDecay} = \frac{\text{movementDuration}}{\text{decayInterval} \times 1000}\,[ms] \tag{1}$$

Then we compute the loss considering each of the packet that the agent is carrying:

$$\text{loss} = N \times \mathit{Distance} \times \text{rewardDecay} \tag{2}$$

And if the loss is higher than the actual reward, the agent wants to deliver.

$$\text{Deliver} \begin{cases} \mathit{yes} & \text{if } \mathit{loss} > R_J - \mathit{Distance} \times \text{rewardDecay} \\ \mathit{no} & \text{otherwise} \end{cases}$$

Where:

- N is the number of parcels that the agent am carrying;
- R_{j} is the reward of the parcel under evaluation;
- *Distance* is the distance from the agent and the parcel in consideration.

Once the agent chooses to deliver, the nearest delivery tile is chosen using the A* algorithm. Although the planner may not select the exact same path, our goal is to prioritize proximity.

3.1.1 Putdown revision

The agent may abandon the putdown intention if better opportunities arise or if conditions change. For example, if a parcel expires while on route, the agent drops that intention.
Another scenario could be that during the delivery, if the agent spots a close and valuable parcel, it will prioritize picking it up.

[4]The maximum carrying parcels can be adjusted by changing the max_carryingParcels hyperparameter.
[5]The distance threshold can be adjusted by changing the radius_distance hyperparameter.

Crowdedness	Variance	Decay	Result
0	0	0	A
0	0	1	B
0	1	0	C
0	1	1	B
1	0	0	A
1	0	1	A
1	1	0	A
1	1	1	A

Table 1: Logic Table for picking up the parcel.

3.2 Pick Up

If the agent decides to not deliver and spot some parcels, it will try to pick them up.
The parcel that the agent is going to choose depends on various metrics, such as the crowdness, the variance and the parcel decay. Table 1 shows the different combinations of those metrics. If a value is 0, it means that the property is absent in the game. For example, the first row (0,0,0) means that the area around the agent is not so much crowded, the spawned parcels have no variance and the parcels are not decaying.

- **A**: This option means to take the safest parcel possible, which is the nearest one. This is because around the agent is very crowded and it decides to be less pretentious. This option is also preferred when there is neither variance nor decay. In this case the reward is always the same, so the best option is to take the nearest parcel.
- **B**: This option chooses the parcel with the highest reward possible when reaching it.
  This strategy is preferred when there is no crowd nearby and the parcel are decaying, so every move will have a cost. The actual value can be calculated using the formula:

$$\text{realReward} = \text{currentReward} - \text{distance} \times \text{rewardDecay}\,[pts] \tag{3}$$

- **C**: Since there is no decay, in this case we consider the absolute value without worrying about the cost of the movement.

When playing a game, luck is also an important factor. Since most of the time we aim to ensure that the nearest parcel is picked when it is crowded, we have introduced a small percentage of the time where, even if it is crowded, the agent can act with a bit of pretentiousness and select a parcel that is optimal (not necessarily the nearest).

$$\text{Best Parcel} \begin{cases} A & \text{if very crowded and } P < 90 \\ B, C & \text{otherwise} \end{cases}$$

3.2.1 Pick up revision

Once a parcel is taken into consideration, a plan is generated to achieve the goal[6]. During the journey from the position of the agent to the location of the parcel, various events can occur, necessitating a revision of the initial intention. Here's some examples:

[6]Details will be discussed in chapter 4.

- if a more appealing parcel is found along the path, our agent might prioritize collecting it first, postponing the initial plan;
- the agent may cancel the intention if the original parcel is taken by another agent;
- if a delivery point is near the current path and the agent is carrying a parcel, the agent might decide to deliver it first.

The described revision of the intention is located in the *revisePickup* function of the *Intention.js* file.

3.3 Target Move

The target move is used when the agent is not carrying a parcel and does not sense any. This intention is used to drive the exploration of the agent. The decision of which tile to target is taken as follows:

- 5% of time the target is chosen randomly among the parcel spawners;
- otherwise, the heat map (section 3.3.1) is taken into consideration.

3.3.1 Heat map

As stated before, the heat map is based on probabilities and the idea behind it is to guide the agent towards the tile that has the highest probability of spawning a parcel. Initially, every spawner has the same probability, calculated as:

$$\mathbb{P} = \frac{1}{number\_of\_parcel\_spawner} \tag{4}$$

Then, we perform an update and a normalization. Specifically, when sensing a new parcel, the corresponding probability of the tile is updated by 5%. By doing so, the total probability would exceed 1 and since we want to keep it fixed, here is where normalization comes into play. First, we calculate the sum of the probabilities; then we set the probability of each tile to:

$$\mathbb{P} = \frac{current\_prob}{total\_prob} \tag{5}$$

The selection of the tile is based on its weight, implemented in this way:

Algorithm 1 Weighted Random Function

```
$cumulativeSum \leftarrow []$
$sum \leftarrow 0$
for all spawner do
    $sum \leftarrow sum + spawner.probability$
    $cumulativeSum.push(sum)$
end for
$random \leftarrow random()$                          ▷ returns a random between 0 and 1
for $i = 0$ to $cumulativeSum.length$ do
    if $cumulativeSum[i] \geq random$ then
        $target \leftarrow cumulativeSum[i]$
        break
    end if
end for
```

With this approach higher weights will "occupy" more numeric spaces, resulting in a higher chance that the random number will fall into the "higher weight numeric bucket".

3.3.2 Target Move Revision

The target move is interrupted when the agent senses a parcel. As it happens, we stop the current plan and a new intention is designated.

4 Plans

Plans are sequences of actions that the agent can perform to achieve an intention. In our project, the plans are managed by the files in the Plan folder, which handle the generation of the problem, the generation of the solution[7] and the execution.
For each intention we have a plan extending the parent class *Plan*, that contains:

- the atomic action that can be performed during the execution such as moving one tile (in any direction), put down a parcel, pick up a parcel;
- other common field relative to all the children classes, such as if the plan has been stopped (*stop*), the *index*, etc.;
- the offline solver function;
- the execution function.

Each child class contains a function to generate the problem of that particular instance, and will solve the problem using one the two planner.

4.1 Execution

Once a sequence of steps is obtained from the planner, the agent can begin the execution. During this process, the agent may encounter obstacles that prevent it from completing the plan. In this case the agent may retry a number of times[8] before replanning. The retry parameter defines how stubborn our agent should be. If the retry number exceed the *max_retry* hyperparameter, then a replan is performed by considering the obstacle. If a new plan is not found, the intention is not achievable and therefore discarded.

4.2 Offline planner

This planner uses the A* algorithm to find a path and - based on the type of plan - will concatenate the final step (put down, pickup).

4.3 PDDL planner

The PDDL planner allow us to solve problem in an automated way. By providing a description of the domain and the problem, this planner generates a plan if it finds one that satisfies the problem constraints. Plans are typically sequences of actions that transform the initial state into the goal state.

[7]There are two types of planner, the Online PDDL planner, and the offline planner that we wrote.
[8]This is not an absolute value but it is a range that can be adjusted by changing the min/max retry hyperparameter.

4.3.1 Domain

The domain provides a description of the set of possible actions, their preconditions and their effects. In our case, we represented the four main movements (up, down, left and right), the pick up and the put down. We also defined the following predicates for the problem generation:

- tile, to define a tile;
- delivery, used to state if a tile is delivery zone or not;
- agent, to specify an agent;
- parcel, to indicate a parcel;
- me, to specify who the agent is;
- at, used to specify that the agent/parcel is on a specific tile;
- right/left/up/down, allows to construct the neighbourhood;
- carrying, if an agent is carrying a parcel.

4.3.2 Problem

The problem description is intended to define the initial state of the world and the goals that the agent has to accomplish.
During the gathering of the information about the map (2.1.1), we also generate the corresponding PDDL representation. In particular, four data structures have been defined:

- *pddlMapObjects*, the set of objects such as tiles, parcels and agents;
- *pddlTiles*, contains all the walkable tiles of the map built as (tile t_x_y);
- *pddlNeighbors*, defines the neighborhood relationship among the tiles as (right t_a_y t_b_y), meaning that t_b_y is on the right of t_a_y;
- *pddlDeliveryPoints*, specifies the tiles which are delivery points as (delivery t_x_y).

Those four data structures are common for the generation of the problem of each plan, but other information is needed to complete the problem:

- pickup: the starting position of the agent, the position of the target parcel;
- putdown: the starting position of the agent, where to deliver the parcel(s);
- targetMove: the starting position of the agent and the destination.

Finally, the goal, which represents the post-condition, must be included.
Once the problem is formulated, it can be paired with the domain to create a plan — a sequence of actions - specified within the domain. As mentioned in the execution section (4.1), the generated plan can fail because we don't take into account the position of the other agents. We don't do that because of the dynamic nature of the environment and even if we try to predict the obstacle, we would lose valuable paths if the prediction is wrong. Instead, we consider the obstacle only when we actually bump into it. If the number of retries exceed a maximum threshold without success, we retain the intention and attempt to achieve it through replanning. The *pddlNeighbors* and *pddlTiles* are temporary modified by removing the tile and its neighbors to reflect the obstacle. In this way the generated plan won't consider paths that use that tile. If no feasible plan is found after these adjustments, it indicates that achieving the intention with the current obstacle is impossible, in such cases we change intention.

5 Communication

The communication part allowed us to coordinate two agents. A set of functions has been implemented to enable the exchange of messages between the two.
More specifically, the communication is initiated thanks to a handshake protocol. As depicted in figure 1, the initiator sends a broadcast message with type *handshake*. As soon as the receiver comes online, it will sense the handshake. Additionally, the body of the message is used to check whether we received the handshake message from our team mate or from another agent, to avoid any coordination error. During this phase we also set an agent role, the sender of the message will take the leading role, meanwhile the other is going to be compliant. After this step, the two agents can start communicating with each other. Even though there is a role splitting, the agents follow a peer-to-peer model, allowing each agent to act according the specific situation and starts a coordination process if needed.

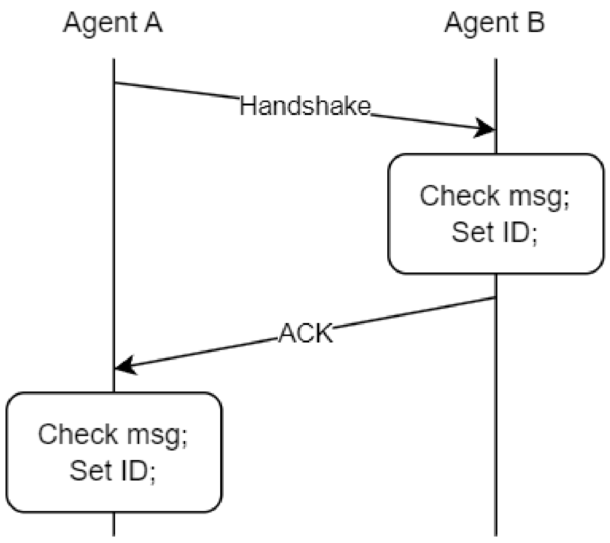

Figure 1: Handshake protocol

5.1 Intention Sharing

To avoid potential collisions between the agents, we implemented a mechanism to exchange the current intention of the agent, so each of them knows what the other intends to do. This knowledge is then used to filter the intention.
More in details, the message contains the type of intention and the target. The first one can be picking up, putting down, targeting and moving; the second one contains information such as the position and, possibly, the id. If the two agents are pursuing the same goal, roles come into play. The leading agent will continue executing the plan, while the compliant agent will stop its execution.

5.2 Beliefs Sharing

To extend the knowledge of the single agent we provided a way to send the current belief to the other agent. The received beliefs are then merged with those related to the agent. This greater awareness about the environment is also used to enrich the intention generation step.

5.3 Coordination Management

The agents can work independently without conflicts, by communicating their intentions, and collaboratively, by sharing its belief in order to have a better understanding of the environment. Most cases don't require coordination because they are handled through revision or replanning. However, some situations disrupt this balance and require stronger collaboration, necessitating synchronization between the two agents. When an agent is blocked by the teammate and there is no other way around, the agent that notices this will start the coordination process. This will stop the current intention of both agents, since it is an emergency. If agent A is trying to deliver a parcel but is blocked by its friend agent B, and there is only one delivery point available, A will initiate a coordination process to swap parcels. A will instruct B to move one tile away and stay still. A will then drop the parcel in the previous location of B and move back. B can now take the parcel and complete the delivery. This process is role independent, so it can also be inverted if the agent A is in the situation of B and vice-versa.

6 Tests and Results

The single agent version has been tested on the nine scenarios present in the first challenge, meanwhile the multi-agent version has been examined on the maps of the second challenge. Both implementations used the PDDL planner and Offline planner to propose a comparison on the scores.
Each test has been executed for 5 minutes using the default set up of the map, with 7 random agents. Unfortunately, the performance of the online planner was not so great due to the network delay, which is something that we can't predict or handle. Since generating a plan took five to ten seconds, maps that have low average and decay would lead our agent to lose the carried parcels while picking up others. To avoid this we set the *max_carrying_parcel* hyperparameter to 1, forcing our agent to deliver after every pickup.
As shown in table 2, despite the intentions of being generated in the same way, the results in terms of points gained are quite different. Again, this gap is caused by the amount of time required by the PDDL solver to return a plan.
The eighth level of the second challenge required heavy coordination, so we increased the average reward of the parcel to 100 since the coordination requires too many steps and the parcel would expire before reaching the delivery (final point 0). Meanwhile for the offline planner we used the standard settings (average reward is 20). Despite that, it has performed much better than the online planner.

Planners	Maps								
	1	2	3	4	5	6	7	8	9
Online PDDL Planner	340	576	609	164	1923	75	67	55	60
Offline A* Planner	1410	2878	2272	1385	4710	651	888	353	4097

Table 2: Results on the maps of the first challenge.

Planners	Maps				
	1	2	3	4	5
Online PDDL Planner	280 + 170	235 + 99	447 + 467	1100 + 1479	466 + 612
A* Planner	1460 + 1010	2175 + 2357	2698 + 2604	8010 + 7899	3358 + 1771

Planners	Maps			
	6	7	8	9
Online PDDL Planner	958 + 663	423 + 324	0 + 572	347 + 160
A* Planner	7096 + 7582	1772 + 2086	0 + 922	4947 + 4387

Table 3: Results on the maps from the second challenge.

7 Additional information

- **radius_distance**: The radius within which an agent can detect and respond to tasks, we can narrow the distance in order to care only the parcel that are within a certain range.
- **crowdedThreshold**: This threshold defines how the agent should behave when encountering multiple parcels.
- **retry**: Parameters defining the number of retries for failed action and is selected a random number between the *min_retry* and *max_retry*. Having a random number instead of a single value helps agents when they bump into; if the number was static and the same for both agent, they would remain stuck indefinitely. With a random number, agents can attempt different approaches over time and potentially become unstuck.
- **blackList**: The blacklist is a structure that contains parcels, delivery points or spawn points to ignore. There are two types: temporary, where no path exists from the position of the agent to the target position due to obstacles (such as other agent, which may move away); permanent, where the target is always unreachable due to the absence of any path even without obstacles.
- **max_carryingParcels**: The maximum number of parcels an agent can carry at one time before delivering.
- **reasonable**: The probability that an agent will make a reasonable decision when presented with multiple options.
- **randomMoveChance**: The probability that an agent will move randomly rather than using the heatmap.

Instead of assigning a specific role, we decided to let the user build different types of agents by changing the hyperparameter in the *Believe.js*. For instance, setting the *reasonable* hyperparameter to a low value and the *randomMoveChance* to a high value would result in an agent primarily driven by luck. To complement this, we could pair it with an agent having a low *radius_distance* and *max_carryingParcels*, creating a more conservative strategy. This would result in a diversified approach to problem-solving.