Unity Prolog Simulation
Autonomous Software Agents - `assignment 2`

Sebastiano Chiari
220527
`sebastiano.chiari@studenti.unitn.it`

1 Introduction

The goal of this project is to build a BDI implementation for a vaccine distribution simulation in UnityProlog. We were provided with a Unity scenario in which vaccine boxes has to be delivered from one house to another through different kind of agents. The implementation of the agents in UnityProlog has to follow an agent-oriented approach.

1.1 Delivery

Inside the delivered zip, along with this report, there are the following elements:

- `KBs` folder, containing all the `.prolog` files
- `Scripts` folder, containing all the modified C# scripts

2 Agents implementation

In the following section, there can be found the implementation and the design choices behind each agent.

2.1 `GameManager` agent

The `GameManager` agent is the one responsible to set up the environment. It creates the boxes and triggers the PickupArea representing the address of the sender.

No changes have been done to the `GameManager` agent with respect to the one delivered with the projects specifications.

2.2 `PickupArea` agent

The `PickupArea` agent represents the post-office box of the house: once it is triggered by the `GameManager` agent, the `PickupArea` agent decides which (not busy) drone must take care of the box it needs to send. Then, it adds a desire to the chosen drone to collect the box and start the delivery process.

When a box is delivered to a `PickupArea`, the drone adds a desire to the `PickupArea` that triggers a plan that checks if the destination of the box (set by the `GameManager`) match with itself.

When a PickupArea gets instantiated through its script, a belief is added stating to which area (`north`, `south`, `east`, `west`) it belongs. This is usefull to correctly design the knowledge base of the other agents, by retrieving the correct intermediate platform to which a drone has to deliver the box.

The case in which the box has the same starting area as its destination was not considered: the box would not even be sent, considering the pickup area as a mailbox of every single house.

2.3 `Drone` agent

Each `Drone` agent has two different types of main plan:

- `shipBox`, this is triggered when the drone is called by a `PickupArea` agent to pick up a spawned vaccine box.
  The drone reaches the caller `PickupArea` and pick up the vaccine box. Then, by checking the box's destination belief, it selects the correct intermediate platform, flies to the landing area and drops down the vaccine box. After, it add another self-desire, `callRailBot`, which is in charge to call the local `RailBot` agent in order to make it collect the vaccine box by adding the desire `collectBox`.
- `deliverBox`, when the drone pick up a vaccine box from a platform to deliver it to its final destination.
  This desire is added to the `Drone` agent by the `RailBot` one. The drone flies to the

platform area, collects the vaccine box and then delivers to the destination after checking the box's destination belief. Once the vaccine box has been delivered to the selected `PickupArea`, it adds to it the `deliver` desire (section 2.2).

Both these plans require as precondition that the drone has a `busy` belief: this belief is added by the agent which calls the drone (e.g., the `PickUpArea` agent or the `RailBot` agent) and it is used in order to avoid the overlapping of different requests at the same time: once a drone is `busy`, no other agent can call it or add anything to it until the drone returns `not busy`.

Since drones has typically a very low battery life, it has been implemented a `recharge` desire, which triggers a recharge plan, after each travel. Once a drone finishes its delivery or shipping plan, it gets added to its knowledge base a `refuel` desire: the drone returns to its charging station and waits for a specific amount of time to be recharged. Then, the `busy` belief gets removed and the drone is ready for a new request. In order to achieve this, it has been modified the C# script for the drone (`Drone.cs`) in order to add a coroutine `WaitForSeconds(float seconds)`, which represents the charging time the drone has to wait. Once the coroutine is over, the drone becomes available and ready to take another delivery plan.

2.4 `RailBot` agent

The `RailBot` agent is responsible for transporting the boxes from the platforms to the exchange area or viceversa and it can only move on its rail. Each `RailBot` has two different types of main plan:

- `collectBox`, after the drone calls the `RailBot`, it gets the box from the platform zone to the exchange area. First, the `RailBot` checks the shipment information. If the box's destination address is in the same area as the starting address, the `sameShippingArea` belief is added to the `RailBot` KB: this triggers the correct `nextStep` plan, which makes the `RailBot` leave the box where it is and call a not busy drone to deliver it to the correct address. Otherwise, the `differentShippingArea` belief is added and the correct `nextStep` plan is activated: the `RailBot` moves the box to the exchange area and calls the `SortingBot`.
- `deliverBox`, invoked by the `SortingBot`, the `RailBot` gets the box from the exchange area and moves it to the correct platform for delivery by checking the box's beliefs. Then, it calls a not busy drone to deliver the box to the correct address.

Given the fact that it is a train, the `RailBot` does not have the need to be recharged. Though, it's implemented a `idle` plan that is activated when the agent is not busy or has no more delivery requests: this plan makes the `RailBot` return to the charging station and wait for new requests.

2.5 `SortingBot` agent

The `SortingBot` is responsible for the sorting of the boxes from an exchange area to another one It is invoked by the RailBot of the starting area and it delegates the management of the box to the RailBot of the destination area. The `RailBot` adds to the `SortingBot` a `sort(Box)` desire, which triggers the plan: the `SortingBot` reaches the box and collects it, then it checks the destination and moves the box to the correct exchange area. Once the box is delivered, it calls the local `RailBot` in order to advance the shipping pipeline.

It is implemented a recharge plan for the `SortingBot` too. As the `RailBot`, once the `SortingBot` has no other duties, the `recharge` plan is activated: it returns to the charging area and waits for new desires.

3 Possible improvements

There can be many ways in which this project can be improved, by taking a deep dive into the C# scripts implementation. I'm going to list some of them below:

- **Vehicles capacity**
  As in assignment 1 different vehicles could have different load capacities, it could be modified how many boxes the `RailBot` or the `SortingBot` can load in order to optimize the deliveries.

- **Sorting bot with battery levels**
  Another modification could be to insert a maximum level of deliveries that the `SortingBot` can manage, before needing a recharge. Then, the charging time could be proportional to the number of deliveries made, and therefore to how much battery has actually been consumed.

- **Intelligent pickup based on distance**
  A possible optimization can be in the way drones are called by the `PickupAreas`. With the current implementation, a `PickupArea` calls a random non busy drone. This could be changed, by making the closest drone be called. This would result in a decrease in delivery times and in an optimization in terms of fuel and also environmental impact. On the other hand, having a uneven distribution of boxes between different `PickupAreas` could lead to a massive use of some drones while others would not even take off despite being always available.

4 Comparison with the PDDL scenario

The UnityProlog scenario is completely different from the PDDL scenario given in the `assignment 1`: in the last, the distribution of the vaccine boxes followed a tree structure, where the boxes started from a center and gradually were distributed to the various sub-centers in a branched way following a precise hierarchy. On the other hand, in this scenario the shape is more like a star: boxes leave from the same points which will later become destinations, they are sent to a centralized distribution center that sorts them correctly and then reroute them to the final destination.

Of course, the structure could be forced, designating a single area as the central point that ships the boxes to all the other areas; however, as is the design of the shipping chain, there would be an overload of some areas (the platforms of the area designated as the central point) and an imbalance between the means of delivery (only one `RailBot` and one `SortingBot` should handle all the load in the first phase) and the flow of the boxes.

Another important aspect to consider is how the decisions are made: in this case, we do not have a centralized approach, where the planner is making all the decisions about what to do, acting as an omniscient agent. The single agents are the very protagonists of the distribution pipeline, adding and removing beliefs and desires according to the BDI method. More, unlike the PDDL scenario, each agent communicate actively with the others, issuing orders, delegating tasks and learning the knowledge base needed to complete the plan without any supervision.

5 Results

The original goal of developing a UnityProlog BDI implementation for a vaccine distribution problem has been achieved. Agents are able to find a solution in order to deliver all the boxes from their starting areas to their destinations according to all the guidelines and constraints specified in the assignment guidelines.