Autonomous Software Agents Report

Dunder Mifflin Paper Company, Inc. Team
Anna Giacomello, Davide Pasetto

July 7, 2023

Abstract

The agent's decision-making process follows a cyclical workflow: The **sensingLoop** function is called periodically, and it generates a list of options based on the agent's observations about the environment surrounding it. The options are filtered based on their utility, which is a custom value based on some calculations, and the best option is selected for execution. The selected option is added to the intention queue, managed by the **IntentionRevisionQueue** class. **IntentionRevisionQueue** continuously revises and executes intentions from the queue, ensuring that the most recent intentions are considered. Each intention is achieved by selecting and executing appropriate plans from the plan library. Plan execution involves carrying out sub-intentions and performing actions to achieve the desired outcome. The process repeats, allowing the agent to adapt its intentions and actions based on the changing environment.
The choice of the queueing method for the Agent represent the choice of making most of the controls over the validity and the goodness of an Intention inside the sensingLoop function. We opted for this implementative option because the execution and revision of Intentions inside the Agent class is done in an asynchronous function, and whilst this allows for a responsive change of Intention based on the surrounding environment, it also caused consistency problems when called several times a second. Therefore we found a more sequential approach to yield the best result.
This implementation, however, caused a lot of overhead in the **sensingLoop** function because of all the controls made. To make the program faster we decided to minimize the number of times the plan is updated to avoid obstacles, by updating the *beliefSet* with eventual new obstacles only when those obstacles actually represents an obstruction on our path towards the goal.

Contents

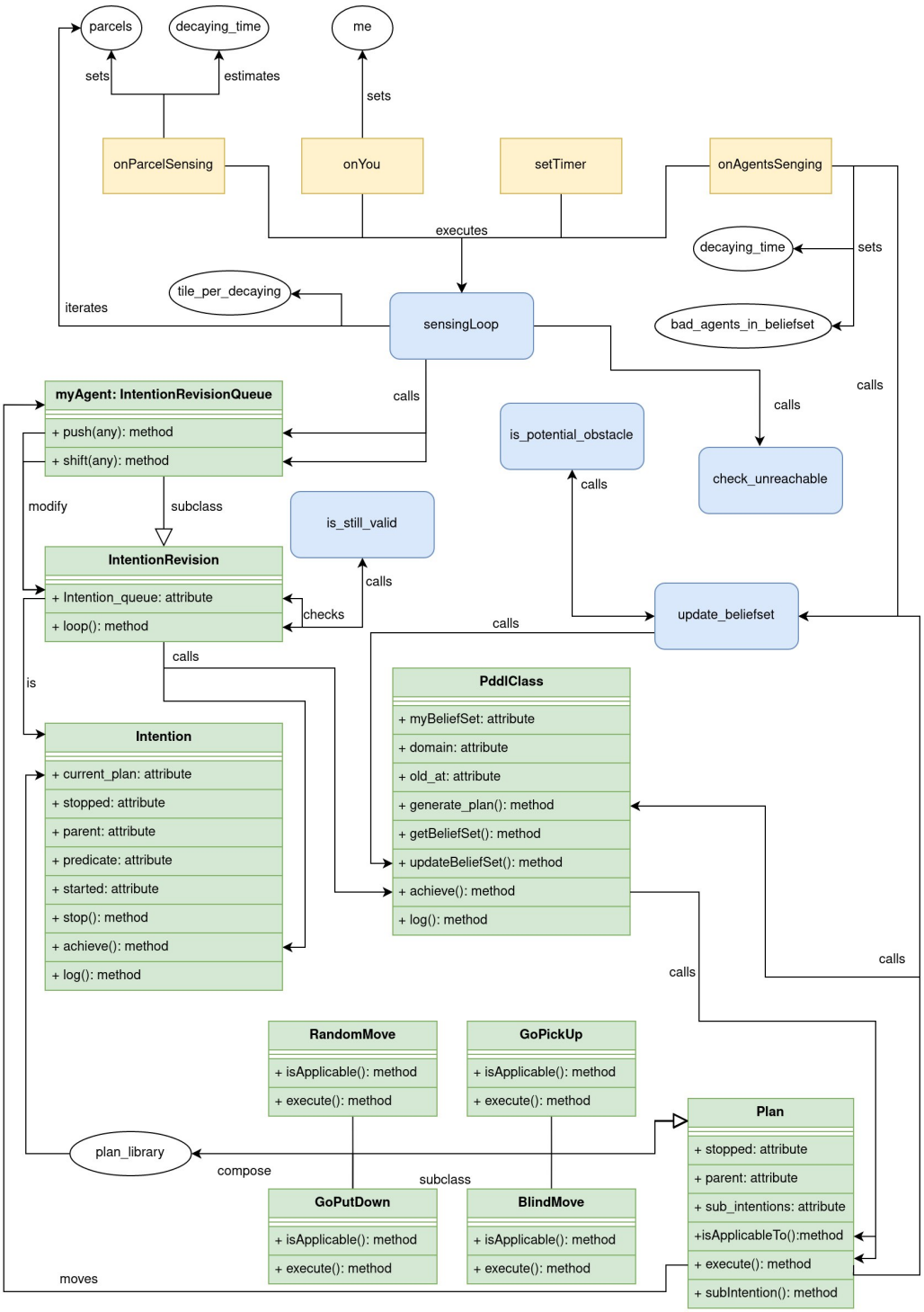
1 UML diagram **3**

2 Agent code **4**
2.1 sensingLoop function . . . 4
2.1.1 Options Generation . . . 4
2.1.2 Options Filtering . . . 4
2.1.3 Execution . . . 5
2.2 IntentionRevision class . . . 5
2.3 IntentionRevisionQueue class . . . 5
2.4 Intention class . . . 5
2.5 Plan class . . . 6
2.6 Plan subclasses . . . 6
2.7 Additional auxiliary functions . . . 6
2.7.1 closest_hole . . . 6
2.7.2 is_still_valid . . . 6
2.7.3 update_beliefset . . . 6
2.7.4 is_potential_obstacle . . . 7
2.7.5 setInterval . . . 7
2.7.6 check_unreachable . . . 7
2.8 Plan Library . . . 7

3 Auxiliary elements **7**
3.1 PDDL class . . . 7
3.1.1 Constructor . . . 8
3.1.2 updateBeliefSet . . . 8
3.1.3 generate_plan . . . 8
3.1.4 getBeliefSet . . . 9
3.2 An important consideration . . . 9
3.3 Map exploration . . . 9
3.4 A* search . . . 9

4 Agent cooperation **10**
4.1 Message Passing . . . 10
4.2 Intention Sharing . . . 10
4.3 Coordination . . . 10
4.4 Conflict Resolution . . . 10
4.5 Synchronization . . . 11
4.6 Map exploration . . . 11

1 UML diagram



2 Agent code

2.1 sensingLoop function

This function represents the main loop where the agent senses its environment, generates options, filters them based on utility, and selects the best option to execute.
It begins by printing the list of friendly agents (friendly_agents) and initializing the tile_per_decaying variable based on time estimations.

2.1.1 Options Generation

The function generates a list of options by iterating over the available parcels. For each parcel, it checks if it is not already carried by another agent and if the target location is not occupied by a bad agent; by 'bad agent' we refer to other (non cooperative) agents that might impede our agent's movement.
If these conditions are met, an option of type *'go_pick_up'* is added to the options list, specifying the parcel's coordinates and ID. If the agent is currently carrying a parcel (checked through *carried_parcels*), an option of type *'go_put_down'* is added to the options list.

2.1.2 Options Filtering

The function then filters the options based on their utility. It calculates the utility for each option by using the following formula:
$total\_gain = (gain - carried\_parcels.size) * current\_d / tile\_per\_decaying$

Let's now analyse all the components to this formula.

- *gain* represents the sum of the rewards of all the parcels currently carried by the agent
- *carried_parcels.size* represents the number of parcels currently carried by the agent
- *current_d* represents the distance from the agent and the parcel it is currently evaluating
- *tile_per_decaying* represents the number of steps the agent has to take for a parcel's reward to decaying by 1. This is calculated by taking the agent's speed into account.

This formula allows the agent to weight its options by taking into considerations multiple factors and their impact on the agent's score. The option with the highest utility is selected as the best option.

2.1.3 Execution

If a best option is found, it is passed to the **myAgent** object to be executed by shifting the intention queue. If no options are available, the agent either goes to a random location or continues its previous intention.

Overall, this function facilitates the decision-making process by generating options, evaluating their utility, and selecting the best option for execution.

2.2 IntentionRevision class

This class handles the intention revision process and manages the intention queue. *intention_queue* is an array that holds the agent's intentions, represented by **Intention** objects. The loop method is an infinite loop that iterates over the intention queue, checks if an intention is still valid, and executes it if it is. If an intention is no longer valid, it is removed from the queue. The loop method also includes a delay using **setImmediate** to allow other processes to run in between iterations. Overall, this class enables the continuous revision and execution of intentions based on the current state of the agent's environment.

2.3 IntentionRevisionQueue class

It was our choice of preference to implement an intention revision queue mechanism. This class extends the **IntentionRevision** class and adds the functionality of queuing intentions in a queue-like manner.
It includes the push method, which adds a new intention to the end of the intention queue. The shift method replaces the current intention in the queue with a new one. If the intention queue is empty, a new intention is added directly using the push method. This class allows for the replacement and ordering of intentions, ensuring that the agent's most recent intentions are considered for execution.

2.4 Intention class

This class represents an individual intention of the agent and handles the execution of plans to achieve the intention. It contains a reference to the parent agent (the **IntentionRevision** object) and a predicate that describes the action to be achieved. The **achieve** method is responsible for executing the intention. It iterates over the plan library, checks if a plan is applicable to the intention's predicate, and executes the plan. The achieve method also handles the stopping of intentions, catches any errors that occur during execution, and removes the intention from the queue once executed. Overall, this class encapsulates the logic for achieving individual intentions by selecting and executing appropriate plans.

2.5 Plan class

This class represents a plan that can be executed to achieve an intention and provides basic functionality for plan execution and stopping. It contains a reference to the parent intention (the **Intention** object) and provides the stop method to stop the plan's execution. This class serves as a base class for specific plan implementations. The plan library contains classes that extend the **Plan** class and implement the isApplicableTo and execute methods specific to their functionality.

2.6 Plan subclasses

The code includes several plan classes, such as **GoPickUp**, **ExploreMove**, **GoPutDown**, and **PlanMove**, which represent specific plans for achieving different types of intentions. Each plan class extends the **Plan** class and implements the **isApplicableTo** method and **execute** methods based on their specific action and conditions. The **isApplicableTo** method checks if the given predicate matches the pattern expected by the plan. The **execute** method carries out the execution of the plan, considering various conditions and potential sub-intentions.

2.7 Additional auxiliary functions

The code includes several additional helper functions that are utilized within the agent's decision-making process.

2.7.1 closest_hole

This function finds the closest hole location to the agent's current position. It iterates over the list of holes, calculates the distance between each hole and the agent, and returns the coordinates of the closest hole that is not occupied by a bad agent.

2.7.2 is_still_valid

This function determines whether an intention is still valid based on its predicate. It checks the validity of different intention types, such as *'go_pick_up'* and *'go_put_down'*, by considering factors like parcel availability and the presence of bad agents.

2.7.3 update_beliefset

This function updates the agent's belief set based on the movements and positions of bad agents. It ensures that the belief set accurately represents the current state of the environment by adding or removing bad agent locations. This function calls the **updateBeliefSet** method from the **PddlClass** class.

2.7.4 is_potential_obstacle

This function checks if a given position is between the agent's current position and a target position. It considers the relative positions of the agent, the target, and the position being checked to determine if it is in the middle or not. The way this gets determined is by considering the "rectangle" that has the agent and the examined parcel as opposite vertices, plus the remaining tiles surrounding the agent. If an agent is located within this area, it is considered to be potentially interfering, and its position is used to update the belief set. The tile is considered to be occupied until our agent gets far enough from it (10 tiles); this is done so the agent can avoid that area for a while, until it is safe to assume that the interfering agent has moved.

2.7.5 setInterval

This statement sets up a recurring timer using **setInterval** to call the **sensingLoop** function every 500 milliseconds. This enables the agent to continuously sense and make decisions based on the updated environment.

2.7.6 check_unreachable

This function check all the parcels saved and deletes the one that are not reachable by the agent to avoid creating intentions impossible to achieve.

2.8 Plan Library

The plan library, represented by the *planLibrary* array, contains different plan classes that implement specific actions and conditions for achieving intentions. The plan classes, such as **GoPickUp**, **ExploreMove**, **GoPutDown**, and **PlanMove**, define the criteria for plan applicability and the execution logic for each plan type. It is worth noting that the "main" class is **PlanMove**, as the other ones eventually call the **execute** function inside **PlanMove**. This function generates a path to go to a certain point, identified by its coordinates, by quering the PDDL online solver. The solver provides a plan, which is then executed by the agent step by step.

3 Auxiliary elements

3.1 PDDL class

The PddlClass class represents an agent's belief system and provides functionality to update the belief set, generate plans based on the beliefs, and retrieve the belief set. It relies on PDDL representations and uses an external API (DeliverooApi) for map information and planning capabilities.
The code creates an instance of DeliverooApi named client using a specific URL and

an access token. It was our choice to instantiate the client here and import it in the agent's code because it was more convenient in order to store information about the agent here.
The PddlClass class represents the belief system and planning capabilities of an agent. It has several private fields:

- *myBeliefSet* of type **Beliefset**
- *domain* of type **PddlDomain**
- *old_at* representing the old location of the agent. This field is needed to de-clutter the belief set each time the agent sets its current position to create a plan. If we didn't remove the old position from the belief set, the belief set would become pointlessly big, potentially causing undesired behavior.

3.1.1 Constructor

The constructor of the PddlClass class builds a PDDL domain by creating four PDDL actions: *right*, *left*, *up*, and *down*. These actions represent the possible movements in a grid-based environment. The PddlDomain object is created with these actions, and its PDDL representation is assigned to the *domain* property.

In the constructor, the belief set is populated based on the map received from the client using the **onMap** event handler. For each tile in the map, a belief named *'valid v[i] v[j]'* is declared or undeclared based on whether the tile is walkable or not. Additionally, increment and decrement beliefs are declared to represent the valid increments and decrements in the X-coordinate and Y-coordinate of the grid.

3.1.2 updateBeliefSet

The **updateBeliefSet** method is used to update the belief set based on the validity of a specific tile identified by its X and Y coordinates. It declares or undeclares the belief *'valid v[x] v[y]'* based on the valid parameter.

3.1.3 generate_plan

The **generate_plan** method is responsible for generating a plan to move from a given location *(x_from, y_from)* to a target location *(x_to, y_to)*. It creates a new belief for the current position, removes the old position belief from the belief set, and sets the new position belief as the old position. Then, it constructs a PDDL problem using the belief set and the goal of reaching the target location. Finally, it calls the **onlineSolver** function passing the PDDL domain and problem, and returns the generated plan as an array of actions.

3.1.4 getBeliefSet

The **getBeliefSet** method returns the entries of the belief set, so that an external caller is able to see what it contains.

3.2 An important consideration

Since we had the necessity to delete facts from the belief set, in order to keep it as tidy and contained as possible, we took the liberty to implement a method, **removeFacts**, in the **Beliefset** class (*Deliveroo.js\node_modules\@unitn-asa\pddl-client\src\Beliefset.js*). The code is the following:

```
removeFacts (fact){
    if (!(typeof fact === 'string'))
        throw('String expected, got ' + typeof fact + ': ' + fact)
    this.#facts.delete(fact);
}
```

3.3 Map exploration

The map **exploration_function** is an auxiliary function that is performed each time the agent isn't able to sense any parcels within its sight range, nor it is carrying anything.
The map exploration algorithm is simple yet gives the opportunity to explore the map in a sufficiently comprehensive way.
The function, with the help of the variable *saved_tiles* is able to keep track of the walkable tiles. In order to choose a tile as the agent's next destination, the function first identifies the map quadrant the agent is currently located into. Then, a point that is reasonably far enough is selected and it is set as the new intention of the agent.
We say 'reasonably' because ideally the point will be located in the opposite quadrant, but a random component is introduced to avoid a deterministic choice for the point, which would result in a repeated behavior by the agent.
The agent keeps exploring until it is able to sense a parcel, which will be then given priority, thus interrupting the exploration.

3.4 A* search

In order to calculate the utility of an option, we need to calculate the length of the path between the agent and the parcels it is currently sensing. We chose not to create an instance of a PDDL problem and call the PDDL solver because we stumbled upon some unexpected behaviors when calling the **method** in this scenario. Instead, we opted for the A* search algorithm, which provides a reliable estimation of the distance between the agent and a parcel, while also keeping into consideration the obstacles along the way.

4 Agent cooperation

At the beginning of the agents' executions, they broadcast a *hello* message. Whoever replies is inserted inside a set of friend agents, called We decided to implement a framework for agent interactions, which includes message passing, intention sharing, coordination, conflict resolution, and synchronization. These mechanisms allow agents to collaborate, share information, and make informed decisions based on the current state of the environment and the intentions of other agents.

4.1 Message Passing

At the beginning of the execution, each agent broadcasts an *hello* message to all the agents in the map. All the agents that reply back are inserted in a set of friendly agents, with which the agent will be able to cooperate. The **client.ask** function is used to send messages to other agents and await their responses through the **reply** function. It allows agents to communicate with each other and exchange information. The messages sent include information about intentions, utility, and queries about agent locations.

4.2 Intention Sharing

Agents can share their intentions with each other using the client.ask function. In the provided code, the **GoPickUp** plan sends a message to other friendly agents to inquire about their intentions and utility related to picking up a specific parcel. This interaction helps avoid conflicts and coordinate actions among agents.

4.3 Coordination

Agents can coordinate their actions by sharing information and making decisions based on that information. For example, the **GoPickUp** plan checks if other agents have a higher utility for picking up a parcel before proceeding with the action. This coordination mechanism helps optimize resource allocation and prevents multiple agents from attempting the same task simultaneously.

4.4 Conflict Resolution

The code includes mechanisms to resolve conflicts and prevent agents from interfering with each other's actions. For instance, if another agent has a higher utility for picking up a parcel, the current agent stops its intention to avoid conflicts. This conflict resolution mechanism helps maintain efficiency and avoid unnecessary competition among agents.

4.5 Synchronization

The code includes a central loop (IntentionRevision.loop()) where agents check their intention queue and execute the next intention if it is still valid. This synchronization ensures that agents take turns and prevent concurrent execution of conflicting intentions.

4.6 Map exploration

When considering the cooperation between two agents, we have to keep in mind that the exploration needs to enforce some additional constraints.
A rule of thumb we decided to follow was that we wanted our agents to be in different quadrants of the map, to minimize the overlap of their respective fields of action.