Autonomous software agents

Luca Zanolo
luca.zanolo@studenti.unitn.it
University of Trento
Trento, Italy

I. Introduction

A. Overview

This report provides an in-depth analysis of the design and implementation of an autonomous software agent tailored for the Deliveroo.js game. The agent is tasked with maximizing points through efficient collection and delivery of parcels in a defined zone. It leverages a Belief-Desire-Intention (BDI) architecture, enabling environmental sensing, belief management, intention formulation and action execution.

B. Approach

My agent employs two distinct strategies for parcel pickup and delivery: one utilizing Breadth-first search (BFS) and the other employing a Planning Domain Definition Language (PDDL) solver available at https://solver.planning.domains:5001/package/dual-bfws-ffparser/solve. These strategies are adjustable to enhance agent performance in diverse environments. The decision-making process follows a BDI loop, using utility values to select the most advantageous next intention.

C. Report Structure

This report is organized into four principal sections. The first section provides a comprehensive overview of a single agent's architecture, detailing the organization of the code, interactions between components and the underlying decision-making logic. The second section offers an in-depth examination of the single agent architecture, outlining the decision-making processes that govern the agent's behavior. The third section expands the single agent architecture to introduce the multi-agent framework, focusing on the mechanisms of communication and coordination among agents and enhancements made to the single agent setup. The last section presents the scores obtained by the agent under different configuration, then the report concludes with a concise guide on deploying the agent in practical.

II. Project Architecture

A. Files and Directories Organization

The project's structure comprises JavaScript files and folders, shown in Figure 1. The `Agent` folder serves as the main directory. Below is a concise description of each class:

1) **`AgentInterface`**: Provides utility methods for the agent's status display and data saving.
2) **`Agent`**: Handles the agent's initialization, movement, pickup, and delivery actions.

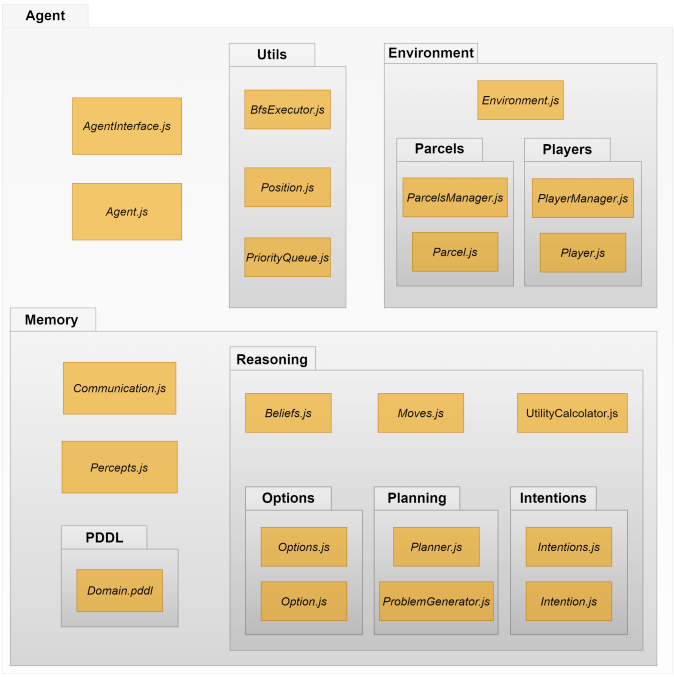

Fig. 1. Project folders and files. The orange rectangle represents files, while the objects in gray scale represent folders.

3) **Utils**: Utility objects for the agent.
   - `BfsExecutor`: Manages BFS-derived actions.
   - `Position`: Manages two-coordinate positions.
   - `PriorityQueue`: Manage entities with a specified priority value.
4) **Environment**: Governs the agent's environmental perception.
   - `Environment.js`: Offers matrix representation and pathfinding methods.
   - **Parcels**: Manages parcels.
     - `ParcelsManager`: Access point for parcels.
     - `Parcel`: Represents individual parcels.
   - **Players**: Manages player entities.
     - `PlayersManager`: Oversees player entities.
     - `Player`: Represents a player.
5) **Memory**: Contains the agent's behavioral logic.
   - `Percepts`: Manages environmental percepts.
   - **PDDL**: Contains the `Domain.pddl` for PDDL strategy.

- **Reasoning**: Problem generation and planning logic.
  - `Beliefs`: Manages agent's beliefs (PDDL strategy).
  - `Moves`: Various movement-related classes.
  - `UtilityCalculator`: Calculates option utilities.
  - **Options**: Generates and filters options.
  - **Planning**: PDDL strategy problem-solving.
    - `Planner`: Finds solutions, manages plan caching.
    - `ProblemGenerator`: Generates PDDL problems.
  - **Intentions**: Manages options and generates intentions.

Subsequent references to these classes will omit the .js extension.

B. Classes Interaction

The interaction among the various classes within the project is managed through two distinct methods.

*1) **Dependency Injection**:* The primary method utilizes the Dependency Injection design pattern, facilitating a modular and decoupled architecture. This approach is particularly evident within the `Agent` class. When creating instances of other classes, such as `ParcelsManager`, `PlayersManager`, `Beliefs`, `Planner`, `Options`, `ProblemGenerator` and `Intention`, the `Agent` instance is injected into them, as shown by the snippet in figure 2. This is accomplished by passing the `Agent` instance (referred to as *this* in the context of the `Agent` class) to the constructors of these classes, ensuring that all components remain accessible through the `Agent` class.

```
this.parcels = new ParcelsManager(this)
this.players = new PlayersManager(this)
this.beliefs = new Beliefs(this)
this.planner = new Planner(this)
this.options = new Options(this)
```

Fig. 2. Code snippet showing initialization of some agent's components with dependency injection.

*2) **EventEmitter Object**:* The second method involves the utilization of the *EventEmitter* object, addressing the need to notify different components under specific circumstances. For instance, when the `Percepts.js` class perceives a new parcel or player, it notifies the corresponding `Player` or `Parcel Manager`. These managers then relay the notification to the `Beliefs` and `Option` classes. In scenarios where a parcel is taken by another entity, they also inform the `Intentions` class. This notification system is crucial, especially considering the asynchronous nature of JavaScript and the use of promises.

Figure 3 illustrates the types of notifications used:

- **movement**: Triggered after each movement.
- **parcels_percept** and **players_percept**: Issued each time a parcel or player is perceived.

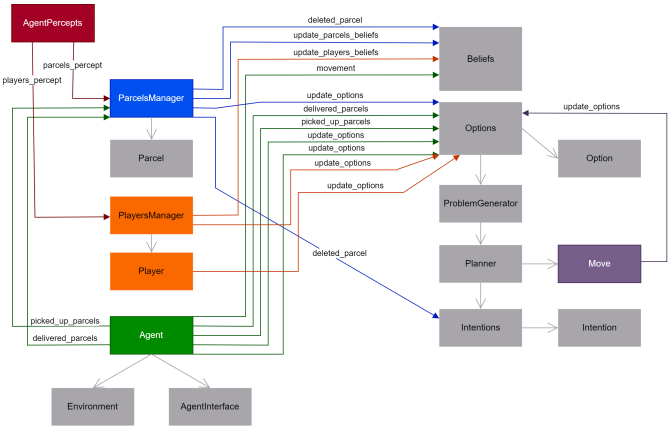

Fig. 3. Interaction of classes through EventEmitter. The diagram illustrates the flow and types of notifications within the system.

- **update_parcels_beliefs** and **update_player_beliefs**: Sent by the managers upon detecting changes in the state of parcels or players.
- **update_options**: Activated to generate new options based on the current state of the agent, also triggered in some other specific situations.
- **deleted_parcel**, **delivered_parcel**, and **picked_up_parcel**: These events aid in managing interactions with parcels, such as when a parcel is taken by another agent or is unreachable.

C. Agent Logic

This section details the flow between components that enable the agent to effectively engage within the game environment. The agent's interaction is based on the schema introduced during the course, adapting this schema to better suit my approach, as depicted in Figure 4.

The agent's interaction with the environment begins with perception through `Percepts.js`, serving as the primary interface. Input is processed and filtered by `ParcelsManager` and `PlayersManager` to determine the presence of new environmental information. Where percepts contain novel information, particularly with the PDDL strategy, this data is propagated to `Beliefs`, as summarized by the yellow entities in Figure 4.

For both PDDL and BFS strategies, novel information triggers an update in options, generating new ones or updating existing ones. Each new option is evaluated for its utility, with zero-utility options being discarded. These options, along with their utilities, are queued in the priority queue managed by the intention revision loop in `Intentions`, prioritizing based on utility values. In the BFS strategy, utility calculation includes precomputing a plan, incorporated into the option. Under the PDDL configuration, plan generation occurs later, with utility estimated using Manhattan distance. If no options are available, a patrolling option is generated for map exploration. The intention revision loop, functional for both strategies,

continually selects the highest utility option, as represented by the orange entities in Figure 4.

Once an option is deliberated, it transforms into an intention. The appropriate plan, corresponding to the option type, is then identified, as shown by the red entities in Figure 4. This plan may be precomputed, newly generated or retrieved from the cache.

The selected intention is then pursued using the corresponding plan. The plan internally manages situations where the path is occupied and, if the plan's target is unreachable, fails, leading to replanning or the selection of the next best option. The plan selected is rapresented by the green entity in Figure 4.

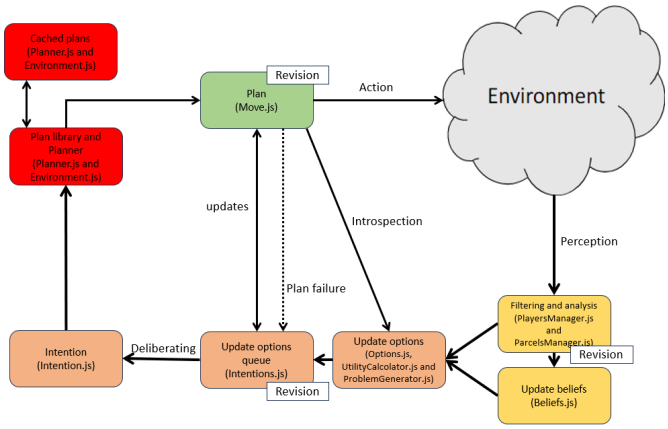

Fig. 4. Revisited BDI logic schema adapted for the adopted approach. It illustrates the modified flow and the main classes involved at each step.

D. Breadth-First Search (BFS) Implementation

The BFS algorithm is a crucial component of the agent, allowing it to find paths in the environment efficiently. The BFS implementation is tailored to navigate through grid-based environments and includes mechanisms for caching and path safety checks.

Algorithm Overview The BFS algorithm starts from a given position and explores all adjacent positions in a breadth-first manner. This approach ensures that the shortest path to the target position is found if one exists.

Caching Mechanism A significant enhancement in BFS implementation is the caching of search paths. Each search result is stored with a unique key representing the start and end positions. The cache is checked at the beginning of each search to determine if a previously computed path can be reused. This reduces computational overhead, especially in static or semi-static environments where paths remain valid for extended periods.

Path Safety Checks Before a cached path is reused, a safety check is performed to ensure that the path remains viable. This is crucial in dynamic environments where obstacles or other agents might block previously valid paths. If the path is deemed unsafe, a fresh search is initiated.

Search Process The search process involves the steps summarized in 5.

BFS Search Process:
Initialize a queue with the starting position and an empty path
while queue is not empty **do**
  Dequeue the front element from the queue
  if element position is the end position **then**
    return path to this position
  end if
  for all valid adjacent positions **do**
    Add position and updated path to the queue
    Store path in cache for future reference
  end for
end while

Fig. 5. Pseudocode for the BFS search implemented.

Failure Handling If the BFS search does not find a path to the target position, the algorithm returns a default response indicating that no path was found, so the agent can decide alternative actions in case of unreachable destinations.

Performance Considerations BFS implementation has been optimized for performance. The caching mechanism significantly reduces the number of searches required, especially in environments where the agent frequently traverses the same paths. During tests, on average, the 50% of BFS searchs performed by the agent find the result inside the cache, thus the search is not performed. Furthermore, each cached search includes a usage counter, and after being utilized twice, it is discarded. Consequently, any subsequent request for that path will necessitate a fresh computation of the plan. This approach guarantees that paths, potentially suboptimal due to the dynamic positions of other players, are not perpetually reused by the agent.

E. PDDL Domain and caching

The last paragraph of this section describe the domain and the caching approach (similar to BFS) adopted for PDDL strategy. The PDDL domain for 'deliveroo.js' agent is designed to rapresent various aspects of Deliveroo.js scenario.

Types: three primary types are defined: 'tile', 'parcel', and 'agent'. These types represent the basic elements in the environment - the grid locations ('tile'), the delivery items ('parcel'), and the delivery agent ('agent').

Predicates: A list of predicates defines the possible states and relationships in the domain. These include predicates to represent the delivery target ('deliveryTile'), the active state of a tile ('active'), the agent's and parcel's location ('at'), the agent carrying a parcel ('carries'), and the relative positioning of tiles ('right', 'down', 'up', 'left').

Actions: The domain features actions that enable the agent to navigate ('move_right', 'move_left', 'move_up', 'move_down'), pick up parcels ('pickup'), and deliver them ('deliver'). Each action is defined with parameters, preconditions, and effects. The preconditions include checks for the

agent's location, the active state of destination tiles, and the absence of other agents at the target location. The effects describe the changes in the agent's location and the state of carrying parcels.

Caching mechanism: A caching system (like for BFS strategy) that stores plan and each sub-plan of the retrieved one from the API is implemented. When a new plan is requested, the cache is inspected to find a match, if founded a call for problem resolution is avoided. Plans are saved in term of starting and ending position, so a plan can be either applied to a pick up, delivery or patrolling action. In case a final action is needed, this is dinamically added.

F. API Changes

We have implemented few modifications to the API used within the project. The modified classes are located in the project folder under the directory `API_changed`. The key changes are explained below:

1) **`BeliefSet:`** Added a function to remove a specified fact. This enhancement addresses the limitation of only being able to undeclare a fact, which previously led to an accumulation of negated facts.
2) **`PddlExecutor:`** Implemented changes in error management. These modifications were necessary to resolve an issue encountered when a movement action detected an interruption and a message is thrown.
3) **`OnlineSolver:`** Aligned the plan format to that used by the previous version of the API, particularly regarding string rapresentation.
4) **`PddlProblem:`** In the `toPddlString` function, we altered the domain name to 'deliveroo', reflecting the name of the used domain. Additionally, the name assigned to the problem now corresponds to the one passed to the `constructor` function. This naming is managed by `ProblemGenerator` class.

III. Single Agent

This section details the functionality of the agent, emphasizing perception, option generation, movement types and the intention revision loop. The agent's interaction with the environment and its mechanisms for generating new options and related utility are described.

A. Agent Initialization

Initialization is the first phase upon agent startup. The Agent class is instantiated with parameters influencing the behavior:

1) **Duration**: Specifies the activity session duration of the agent. Optional, but when set, it enables mechanisms that can use the remaining time.
2) **moveType**: Defines the strategy, either 'PDDL' for PDDL strategy or 'BFS' for BFS strategy.
3) **fastPick**: A boolean parameter allowing the agent to quickly pick up adjacent parcels.
4) **changingRisk**: A value in the range [0,1], used when comparing the utility of a new option against the current plan (see Algorithm 1. This parameter modulates the agent's tendency to switch plans.
5) **adjMovementCostWindow**: Used by `UtilityCalcolator` to set the interval of update for the movement penality parameter, used to compute actions utility.
6) **disappearedPlayerValidity**: The duration in milliseconds before an agent is considered missing due to the absence of perceptual updates.
7) **playerCheckInterval**: The frequency in milliseconds at which player statuses are re-evaluated.

Algorithm 1 Use of changing risk in evaluating new options against the actual.

```
if CurrentOption.utility < (NewOption.utility * changingRisk) then
    Stop current intention and start the new best one
end if
```

The initialization connects the agent to the server via DeliverooApi, incorporates an event emitter (detailed in II-B2), and utilizes `Parcepts` as first interface for server information processing. Components loaded include all the classes specified in II. Notably:

The two component described below will manage all the information of the environment for the agent.

1) **Environment:** The `Environment` class is initialized with a matrix representation of the environment. This matrix categorizes the tiles into accessible, inaccessible and delivery tiles, represented by the numbers 1, 0 and 2 respectively. The initialized environment matrix is essential for setting up the `Beliefs` component, particularly in defining the navigable space for the agent.
2) **Beliefs:** In the context of the PDDL strategy, the `Beliefs` class represents the agent's understanding of the world. During initialization, it incorporates all possible movements within the environment and its own state. Each movement possibility is defined by predicates 'right', 'left', 'up', and 'down'. For instance, a predicate like 'right t1_2 t2_2' indicates that a rightward movement from tile 't1_2' leads to 't2_2'.

Following the synchronization of these components, they become operational and start managing various events and updating themself. This activation marks the commencement of the intention revision loop. The logic and sequence of this initialization process are encapsulated in the `start()` function of the `Agent` class.

B. Percept information management

This section describe how the information workflow from the environment is processed, analyzed and saved by the agent.

1) Percepts and Managers: Right after the initialization of the agent, `Parcepts` starts to manage the percepts from the environment. This class will emit an event based on the perception type. When parcels are perceived, a `parcels_percept` is emitted, while for player a `players_percept` is emitted

(see figure 3). In both cases, the event is captured by the correspondant manager.

1) **Players manager**: when players are managed they are compared to the actual players and related information already available. For each player is checked if already encountered, in this case we will update the istance of Player for that player, if the position of the player is the same, nothing change, otherwise we update the position of the player and the related history of positions. In the case in which a new player is encountered an instance of Player is created with the actual perceived information. Thus in case an already encountered player is updated or a new is detected, then this manager will notify the Beliefs class (only with PDDL strategy) and then the Options. This class includes a timer set to **playerCheckInterval** ms, initiating an evaluation of the most recent percept received for each player. If a player lack updates for **disappearedPlayerValidity** ms, it is marked as disappeared, triggering updates to beliefs and options.
2) **Parcels manager**: When parcels are perceived for each never seen parcel a new istance of Parcel is created and the related reward timer started. For not new parcels, the details of the already existing instances are updated. For each single parcel is possible to use the methods `isTaken`, `isMine` and `isAccessible` to determine its state. The parcels are managed using three group:
   a) **Parcels**: a Map that contains for each parcel id the correspondant object.
   b) **My parcels**: a Set that contains all the ids of the carried parcels.
   c) **Deleted Parcels**: a Set that contains all the ids of the deleted parcels; ids inside this set will not be considered anymore and the addition to is irreversible.

   The manager, similarly to the one for player, will notify an update to the belief set only if there are some changes in the parcels states or if there is a new parcel.

C. Types of Options

In the system, agent options are generated by the Options class in response to these events: `update_options`, `picked_up_parcels`, `delivered_parcels` or `movement`. The two types of options are `BfsOption` and `PddlOption`, both extending from a superclass `Option`. Each of the options generated is responsible to computing utility and retrieving plans through their update methods.

1) Option (Superclass): `Option` is directly used for a first definition of patrolling option, generated by selecting one random valid position of the map as target. When patrolling will be then encapsulated in an intention, it is mapped to either `PddlOption` or `BfsOption`, based on the chosen strategy. In this way, also the patrolling behaviour will contribute to the cache of the agent.

2) BfsOption and PddlOption: Both `BfsOption` and `PddlOption` are structured to manage three types of agent behaviors: delivery, pickup and patrolling. `BfsOption` is created with a unique identifier and an optional parcel. Its update method revises the plan, utility and final position according to the action type and the current environment state. `PddlOption` is similar to `BfsOption` but includes asynchronous updating of the plan and utility. PddlOption utility is refined in two steps: initially it is an estimation since the plan that indicates the exact number of steps to perform is not available. Then, when `Update` function is called to retrieve the plan, the utility is updated with the new plan information, this result in a more accurate utility for the option.

D. Utility Calculation

The `UtilityCalculator` class plays a foundamental role in calculating option's utility, for both the approaches. This section outlines how the utility is computed, starting from the general idea, then are presented mechanism to address specific situations.

1) General Approach: The utility of picking up a parcel is an estimate of the reward the agent will achieve by taking and immediately delivering the parcel. For both BFS and PDDL strategies, we have distinct methods to calculate pick up and delivery utilities, but the basic algorithm is the same.

- **Pick Up Utility** (Algorithm 2): Calculates utility based on the actual reward, carried parcels and costs associated with pickup and delivery distances, adjusted by the movement penalty.

Algorithm 2 Pick up utility - General

function PICKUPUTILITY($startPos$, $parcel$)
  $actualReward \leftarrow$ GETMYPARCELSREWARD
  $carriedParcels \leftarrow$ CARRIEDPARCELS
  $search \leftarrow$ GETSHORTESTPATH($startPos, parcel$)
  if LENGTH($search.path$) $= 0$ **then**
    return $\{value : 0, search : search\}$
  end if
  $pickupDistance \leftarrow$ LENGTH($search.path$)
  $pickupCost \leftarrow pickupDistance \times movementPenality$
  $deliveryDistance \leftarrow$ LENGTH($parcel.pathToDelivery$)
  $deliveryCost \leftarrow deliveryDistance \times movementPenality$
  $cost \leftarrow pickupCost + deliveryCost$
  $utility \leftarrow (actualReward + parcel.reward) - (cost \times (carriedParcels + 1))$
  return $\{value : utility, search : search\}$
end function

- **Delivery Utility** (Algorithm 3): Focuses on the actual reward adjusted by a factor based on the number of carried parcels and the delivery cost.

Algorithm 3 Delivery utility - General

```
1: function PICKUPUTILITY(startPos, parcel)
2:     actualReward ← GETMYPARCELSREWARD
3:     carriedParcels ← CARRIEDPARCELS
4:     search ← GETSHORTESTPATH(startPos, parcel)
5:     if LENGTH(search.path) = 0 then
6:         return {value : 0, search : search}
7:     end if
8:     if carriedParcels == maxParcels then
9:         return {value : Infinity, search : search}
10:    end if
11:    deliveryDistance ← LENGTH(search.path)
12:    deliveryCost ← deliveryDistance × movementPenality
13:    cost ← pickupCost + deliveryCost
14:    carriedParcelsFactor ← 1 + (carriedParcels/maxParcels)
15:    utility ← (actualReward * carriedParcelsFactor) − (cost × carriedParcels)
16:    return {value : utility, search : search}
17: end function
```

2) Special Configurations: All the mechanism describe here modify in different point the two procedures presented above.

- **Infinite PARCEL_DECADING_INTERVAL** (Algorithm 4): The utility is simplified to consider only distances, as the movement penalty becomes fixed.

Algorithm 4 Utility for both strategies - Variant when PARCEL_DECADING_INTERVAL is infinite for both 2 and 3.

```
1: function PICKUPUTILITY(startPos, parcel)
2:     search ← GETSHORTESTPATH(startPos, parcel)
3:     if LENGTH(search.path) = 0 then
4:         return {value : 0, search : search}
5:     end if
6:     pickupDistance ← LENGTH(search.path)
7:     utility ← 1 + (1/(pickupDistance))
8:     return {value : utility, search : search}
9: end function
```

- **PDDL Strategy** (Algorithm 5): For both pick up and delivery, distances are estimated using Manhattan distance when direct paths are unavailable, this situation verify each time a new PdllOption is generated. Then, when the plan is computed the utility is updated using the same function, but passing the length of the plan as distance to the function, thus, the function will not use the estimated one.
- **MAX_PARCELS Constraint** (Algorithm 3): Triggers a delivery with infinite utility when the agent reaches its maximum parcel carrying capacity.
- **Time Constraint** (Algorithm 6): Applies a check based on the specified agent duration, calculating the time required for delivery actions as the distance multiplied by the average movement time, with an additional factor for safety.

Algorithm 5 Pickup and delivery utility with PDDL strategy - Addition to algorithms 2 and 3 for PDDL strategy.

```
1: function PICKUPUTILITY(startPos, parcel, distance = null)
2:     ...
3:     pickupDistance ← MANHATTANDISTANCE(startPos, parcel)
4:     if distance is not null then
5:         pickupDistance ← distance
6:     end if
7:     pickupCost ← pickupDistance × movementPenality
8:     ...
9: end function
```

Algorithm 6 Delivery utility if duration parameter (III-A) is available - Addition to algorithm 3

```
1: function DELIVERYUTILITY(startPos, parcel, distance = null)
2:     ...
3:     remainingTime ← agent.duration − elapsedTime
4:     if remainingTime < deliveryDistance * (movementTime * 1.5) then
5:         return {value : Infinity, search : search}
6:     end if
7:     ...
8: end function
```

This utility calculation is central to the intention revision loop, where options with higher utilities are prioritized, guiding the agent's behavior in various scenarios.

E. Intentions

Having discussed how the agent initializes, manages percepts and generates options, we now focus on the transformation of these options into intentions.

Options, once generated and sorted by utility in the `Options` class, are pushed to the `Intentions` class. This class, as detailed in Section II-B, contains the main operational loop of the agent. Initiated during the agent's startup, this loop continuously selects the highest utility option, validates it and encapsulates it into an executable intention. Validation involves ensuring the option's consistency, such as verifying parcel availability for pickup options and if a delivery option has sense.

When a new list of options is pushed to `Intentions`, these options could either be updates to existing ones in the queue or entirely new additions. Existing options are updated with new details, while new options are added to the queue. Each incoming option's utility is compared with the currently executing option, leveraging the risk adjustment mechanism outlined in Section 4. If a new option presents higher utility

than the one in execution, this option is the first one of the pushed options. This prioritization is ensured by the initial sorting of new options, meaning if the first option examined does not surpass the current one in utility, no subsequent new options will either. This new best option is pushed into queue and the current one is stopped. The new best option is then executed in the next iteration.

A specific event handled within `Intentions` is the `deleted_parcel` event, introduced in Section II-B2 and raised exclusively by `ParcelManager`. This event is triggered when a parcel is delivered or becomes unreachable. Upon this event, `Intentions` removes the corresponding option from its queue, if present, to prevent the execution of now irrelevant options. This mechanism was implemented to address situations where an option, despite losing its validity, remains queued for execution. It ensures that the options in `Intentions` closely reflect the current state of the environment.

F. Fast delivery or pickup

The `ActualTileCheck` function is called for most of the movements, examining the agent's current tile at each step. If a parcel is present, it is picked up immediately, and if appropriate for delivery, the delivery is executed. The `fastPick` parameter, when enabled, extends this mechanism to also check adjacent tiles for parcels. Detection of a parcel on an adjacent tile triggers an immediate move and pickup, bypassing updates to other components, as the agent will resume its planned path. If the fast pickup occurs on the next planned tile, the agent simply continues along the path without returning to the previous tile, effectively skipping the planned next movement.

G. Environment Interaction

1) Plan Execution: Once an option is selected in `Intentions`, it is encapsulated into an intention, processed, and executed. The processing step involves determining the appropriate type of move for the intention, handled by the `Intention` class. This process leads to the execution of the intention, which follows one of two movement types: `BreadthFirstSearchMove` for BFS strategy and `PddlMove` for PDDL strategy.

The following steps describe the move execution approach used from both the strategies.

1) **Step 1:** The execution begins with a validation of the plan within the intention, examining its feasibility and alignment with the agent's current position.
2) **Step 2:** The plan is executed using `BfsExecutor` for BFS strategy and `PddlExecutor` (from the provided API) for PDDL strategy.
3) **Step 3:** Throughout the execution, the agent performs checks for external interruption of the current intention, inspects the next visible tiles along the path and executes a special control, `ActualTileCheck`, before each movement.

If a movement fails, it is retried twice before triggering an error and halting the intention. If a path's tile is occupied, the plan is immediately revised and a new path is generated, with a corresponding update in the intention's utility due to possible changes in path length. If a goal become unreachable, indicated by the lack of a valid plan update, the intention is terminated, since the the goal is not reachable.

IV. Multi-agent Implementation

This section highlights the extensions made to the single-agent architecture to enable multi-agent cooperation. The multi-agent framework is built upon the single-agent structure, enhanced with a communication module and a team manager. Three strategies are employed: two involving agents using either BFS or PDDL strategies, managed by `Communication` and `CommunicationForBfs` or `CommunicationForPddl`, respectively, and a third strategy extending the PDDL approach to include multi-agent planning capabilities.

A. Added Classes

The multi-agent system introduces four new classes, in Figure 6, to facilitate agent communication and team management:

- `TeamManager.js`: Manages team-related information and validates team member intentions within communication classes.
- `Communication`: Serves as a high-level communication module, synchronizing the team, managing events and coordinating parcel sharing.
- `CommunicationWithBfs`: Specializes in BFS strategy, filtering messages and extending `Communication`.
- `CommunicationWithPddl`: Designed for PDDL strategy, it also supports the multi-agent planning with strategy 3, handling synchronization and plan establishment.

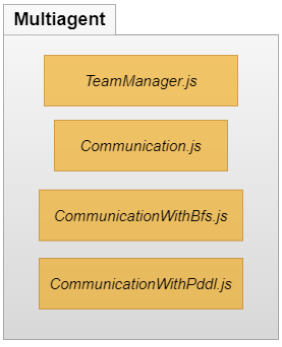

Fig. 6. Project multi-agent module folder.

B. Added Events

In the multi-agent setup, additional events complement the existing single-agent events to facilitate team member interactions, as shown in Figure 7. These events are:

- **left_parcel**: Triggered when an agent leaves parcels on a tile for teammate sharing, informing the `ParcelManager` of this action.
- **parcels_percept_from_team**: Occurs when a teammate shares perceptions of parcels, which are then processed by the `ParcelManager`.
- **players_percept_from_team**: Similar to parcel perceptions, this event conveys teammate's player perceptions to the `PlayerManager`.
- **Communicate_delivery**: Signals that an agent has delivered parcels, facilitating team alignment.
- **Communicate_pickup**: Announces that an agent has picked up parcels, keeping the team informed.
- **new_player_info**: Shares updates on new or changed player information between agents.
- **new_parcels_info**: Exchanges updates on new or altered parcel information between agents.

These events, specifically designed for multi-agent coordination, are activated only when the multi-agent mode is enabled. Some other addition and changes are deployed inside various classes to explicitly manage multi-agent specific states, they can be easily identified by the `if(multiagent)` conditions in the code.

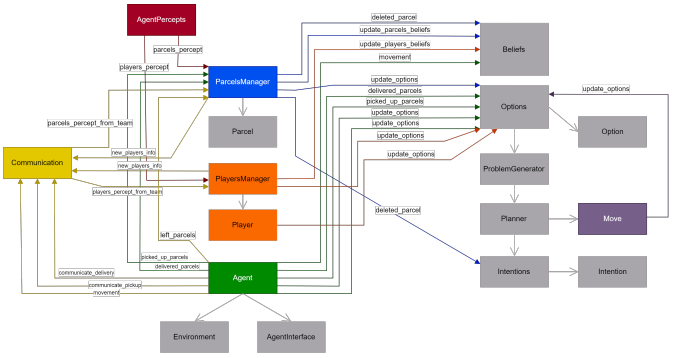

Fig. 7. Class interactions through EventEmitter, highlighting new events for multi-agent configuration in yellow.

C. Synchronization Procedure

The synchronization process ensures all agents are synchronized before starting their operational loop. Figure 8 shows the sequence diagram for the synchronization procedure.

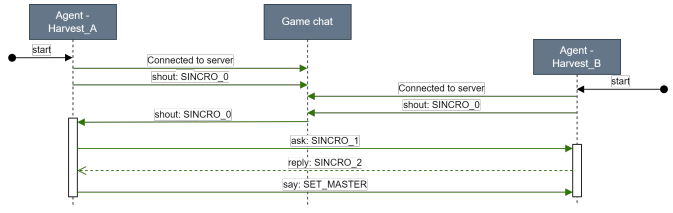

Fig. 8. Interaction of classes through EventEmitter. The yellow object are the events added to 3 for multi-agent configuration.

The steps are as follows:

1) An agent connects to the Deliveroo system and broadcasts a `SINCRO_0` message, signaling its presence.
2) Upon receiving a `SINCRO_0` message, any pre-connected agent initiates the synchronization process (agents connecting subsequently will also send this broadcast and the other agents can trigger the synchronization)
3) The detecting agent sends a `SINCRO_1` message to the newly connected agent, provided the sender is recognized.
4) Upon receiving `SINCRO_1`, the recipient verifies the sender's identity, adds the sender to its team, and sends a `SINCRO_2` message in acknowledgment.
5) The pre-connected agent awaits a `SINCRO_2` response, which upon receipt, confirms the addition of the responding agent to the team.
6) Master determination is based on the count of `SINCRO_0` messages received. The first agent, having received `SINCRO_0` messages equal to the team size minus one, is set as the master.
7) The identified master then broadcasts a `SET_MASTER` message, asserting its role and concluding the synchronization process.

Following the `SET_MASTER` acknowledgment, all agents proceed with their internal loops.

D. Sharing Parcels Mechanism

The parcel sharing mechanism, applicable to the first two strategies, allows agents to assist the teammate in delivering parcels it cannot deliver. This coordination process is presented in sequence diagram of Figure 9 and initiates with an `IMPOSSIBLE_DELIVERY` message from an agent who cannot complete a delivery.

The recipient agent evaluates whether assisting yields higher utility compared to its current task. A positive assessment leads to sending a `DELIVERFORYOU_0` message to commence cooperation, while a negative one results in a `NOACK` message, indicating inability to assist.

Upon agreeing to cooperate, the assisting agent formulates a `ShareParcelsOption` and determining a midway meeting point through bidirectional search. This meeting point is communicated back and option pushed into intention queue with infinity utility. Upon receiving or sending the communication, each agent halts its current intention and, given the sharing intention's utility is set to infinity, it is immediately executed for them both.

The agents then proceed towards the meeting point. When midway tile is reached or a failure is encoutered and they are adjacent, an alignment phase starts, managed by `FirstAlignment` function. Each agent independently ensures alignment, they are aligned when both of the agents have received the `ALIGNMENT_1` message from the other agent.

Subsequent actions depend on the agent's role:

1) **Leave:** The agent in possession of the parcel acts first by depositing it at the current location. Subsequently,

the agent moves in a direction opposite to the previous move, ensuring it remains within a valid position. This action concludes the second alignment phase for the teammate by sending an `PARCELS_LEFT` message. The agent then awaits a confirmation message from the partner, marking the termination of the parcel-sharing phase.

2) **Take:** The agent assigned to retrieve the parcel must await the partner's action of leaving the parcel. This waiting phase involves monitoring the `parcelsState` within `ShareParcelsOption` for a transition from 'taken' to 'left', a change that solely the partner can trigger. Upon the state changing to 'left', after `PARCELS_LEFT` message, the agent proceeds to the adjacent tile to collect the parcels. Following successful pickup, a (`SHARINGCOMPLETE`) message is sended to the teammate, signaling the completion of the sharing interaction.

If in any moment of this process a problem arise, the agent that detect the problem will send a `PARCELSHARINGINTERRUPTED` message to the other agent, to handle correctly the termination of the cooperation.

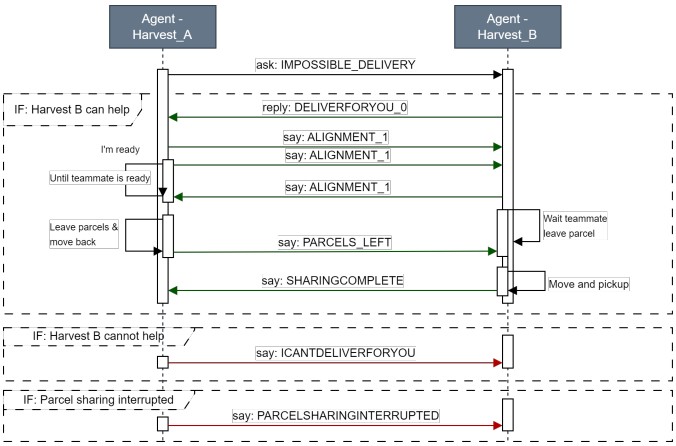

Fig. 9. Sequence diagram illustrating the parcel sharing mechanism between agents.

E. Agent Cooperation with Strategies 1 and 2

In the first two strategies, a pair of agents, either both employing BFS or PDDL, collaborate closely. This cooperation leverages the parcel sharing mechanism and an intention validation system to ensure exclusive intentions. Despite operating individually, agents share environmental perceptions, allowing each to independently identify options based on both their own and their teammate's observations.

Communication between agents is efficient; not every percept triggers a message. Instead, as detailed in Section III-B for single agents, communications are reserved for significant changes in parcel or player states. This selective messaging ensures only pertinent updates are exchanged, maintaining focus on relevant environmental changes.

A master-slave hierarchy is used in the intention validation mechanism. Only the slave agent seeks intention approval from the master, who autonomously validates its own intentions. This arrangement prevents the master from incurring delays due to validation requests, positioning the slave to rely on the master's decisions.

Both of this strategies use the `multiagentLoop` function of `Intentions` class.

F. Agent Cooperation with Strategy 3

Strategy 3 introduces a novel approach where the master agent computes a multiagent plan and shares it with the slave. This strategy is managed through the `multiagentPddlLoop` method in the `Intentions` class. The key steps, applicable to both the master (M) and slave (S) agents, for computing a multi-agent option, are outlined as follows:

1) **Initialization**: The intention loop begins, and agents check for a defined `multiagentOption` within the `Communication` class.
2) **Plan Proposal (M)**: Absence of a `multiagentOption` triggers the master to consider a new multiagent plan. It analyzes its intention queue and signals readiness for multiagent planning by sending a `READYFORMULTIAGENTPLAN` message to the slave.
3) **Response Handling (S)**: Upon receiving the master's message, the slave assesses its ability to participate in the multiagent plan:
   - If currently engaged in a multiagent task, the slave declines with a `NOACK` message.
   - Otherwise, the slave evaluates its options and the master's sent ones, agreeing to cooperate if there are two or more options available. It then generates a `ShareParcelsOption` and temporarily halts its current intention.
4) **Plan Coordination**:
   - The slave communicates its agreement and details (position and options) back to the master using an `ACK` message. Immediately afterward, it sends an `ask` message to specify the reply address to receive the new multiagent option from the master.
   - The master, upon receiving the slave's consent and details, updates the slave's position, incorporates the shared options, and awaits the specific message that contains the reply object.
5) **Multiagent Plan Computation (M)**: The master computes the multiagent plan, aiming to maximize utility considering all the options available. Success leads to sharing the new option with the slave; failure results in notifying the slave of the inability to generate a plan.
6) **Execution**: Both agents execute their assigned parts of the plan. The master continues its tasks independently once its portion of the plan is complete, monitoring the slave's progress.
7) **Completion and Reset (S)**: Upon finishing its tasks, the slave notifies the master, resetting their

`multiagentOption` to signal readiness for future cooperation. The master, upon acknowledgment from the slave, also resets its state, and the cycle can restart.

V. Tests

This section presents the performance of the single agent and multi-agent configurations. The measures used in the tables include the score obtained by the agent, the total score divided by the time period (Score Sec.) and the average time for each agent move. The scores obtained by the agent under various configurations across seven testing environments (`challenge_21`, `challenge_22`, `challenge_23`, `challenge_24`, `challenge_31`, `challenge_32`, and `challenge_33`) are presented. The single agent was tested with different parameter configurations, with each test having a duration of 5 minutes and performed twice for each environment. The multi-agent tests were executed twice for each environment and had a duration of 3 minutes.

The scores presented in Tables I and II represent the best score obtained with the specified strategy in the specified environment. For a detailed inspection of the test results, the `Analysis.ipynb` Python notebook in `\test\performance_analysis` can be used. Additionally, the scores obtained with different configurations for both single and multi-agent setups are summarized in the Excel file `test_scores` in `\test\performance_analysis`.

It is important to note that during testing, the API and planner initially used for designing the agent underwent significant updates. These updates resulted in an increased delay in plan retrieval compared to the initial version, averaging 3500ms with the remote planning service and 1500ms with a Docker local instance compared to the previous 200-300ms delay with a remote instance. Consequently, the performance of the PDDL strategy was significantly impacted, leading to much lower scores compared to the BFS strategy.

Environment	Strategy	Score	Score Sec.	Time per Move (ms)
challenge_21	BFS	510	1.7	668
challenge_21	PDDL	270	0.9	1694
challenge_22	BFS	800	2.66	233
challenge_22	PDDL	57	0.19	1214
challenge_23	BFS	4456	14.85	227
challenge_23	PDDL	294	3.04	1000
challenge_24	BFS	1793	5.97	209
challenge_24	PDDL	513	1.71	486

TABLE I
SINGLE AGENT PERFORMANCE IN TESTING ENVIRONMENTS.

Environment	Strategy	Score	Avg. Score Sec.
challenge_31	BFS	1427	3.96
challenge_31	PDDL	550	1.52
challenge_31	PDDL_1	223	0.61
challenge_32	BFS	2542	7.06
challenge_32	PDDL	336	0.93
challenge_33	BFS	846	2.35
challenge_33	PDDL	277	0.76
challenge_33	PDDL_1	202	0.78

TABLE II
MULTI-AGENT PERFORMANCE IN TESTING ENVIRONMENTS.

VI. Evaluation

In the evaluation of my agent within the Deliveroo.js game environment, I observed notable performance differences between the BFS and PDDL based strategies. The BFS strategy consistently achieved higher scores across all test environments, demonstrating its efficiency and effectiveness in real-time decision-making and task execution. On the other hand, the PDDL strategy, while offering a more sophisticated planning capability, was penalized by longer planning times and lower overall scores.

The multi-agent tests further highlighted the potential of collaborative efforts in enhancing task performance, again with BFS multi-agent configuration outperforming the PDDL counterparts. However, the complexity of coordinating actions and sharing intentions among agents introduced additional challenges, particularly in the synchronization of multi-agent plans.

VII. How to Use the Agent

Outlined below are the steps to deploy and run the agent effectively:

1) **Update API:** Ensure that the API incorporates the modifications detailed in Section II-F to align with the agent's requirements.
2) **Configure Settings:** Access `Config.js` to adjust the session parameters.
3) **Launching the Agent:**
   - *Single Agent:* To initiate a single agent session, execute `node index.js`.
   - *Multi-Agent:* For a multi-agent setup, launch each agent individually using `node index.js [agentIndex]`, where `[agentIndex]` is a unique identifier for each agent. The available identifiers are 1 and 2, the first of them that connect to server, will be the leader of the team.