Single and multi-agent solutions for handling a parcel-delivery game

Marco Gandolfi, marco.gandolfi-1@unitn.it, 258017
Seyed Morteza Mojtabavi, seyed.mojtabavi@unitn.it, 256328

Date: July 13, 2025

University of Trento
Department of Information Engineering and Computer Science
Via Sommarive 9, 80123
Povo (TN), Italy

1 Introduction

The following report outlines the core components of our approach to developing an autonomous software agent system designed to compete in the Deliveroo.js game. In this game, players navigate a grid-based map to collect packages of varying value and deliver them to designated cells, aiming to maximize their individual scores by optimizing pickup and delivery actions, while also accounting for the dynamic nature of the environment; multiple agents operate simultaneously on the same map, and they may compete for the same packages or delivery spots.

We begin by presenting a single-agent system based on the Belief-Desire-Intention (BDI) architecture, focusing on the specific strategies employed to implement each stage of this framework. This single-agent system serves as a baseline for more advanced approaches.

First, we introduce a multi-agent system in which two agents collaborate to achieve higher scores by sharing information about the environment and their respective intentions. Subsequently, we explore an attempt at the integration of an online planner into the agent's behavior, enabling it to generate action plans using the PDDL (Planning Domain Definition Language) standard.

2 Belief set representation in the single-agent system

At all times, the agent maintains a belief set, which consists of a collection of assumptions about the environment. This set is continuously updated on the basis of new information obtained through the agent's sensing activity. For clarity and modularity, the belief set is divided into several maps, each dedicated to tracking a specific type of entity. In this report, we focus in particular on the representation of the environment map, the packages, and the other agents currently active in the game.

2.1 Map representation

At the beginning of the process, the agent receives data describing the map structure, including the width, height, and the `tiles` array. The `tiles` array specifies, for each tile on the map, its coordinates and type. This information is used to construct three key components that make up the agent's internal representation of the map:

- `deliveryCells`: a map containing all cells whose type is equal to 2. These cells represent delivery zones, where agents can place carried packages to add their respective rewards to the total score. This set is later used to determine valid destinations for parcel delivery actions;

- `generatingCells`: a map containing all cells whose type is equal to 1. These are the cells in which packages are expected to be generated and, therefore, identify the tiles our agent should consider exploring in order to find new parcels;

- `graph`: a representation of the entire map as an undirected graph, where each node corresponds to a specific tile and edges connect adjacent tiles. This structure is later used to compute the shortest path to a given location, and can be dynamically updated by temporarily marking certain nodes as non-traversable due to the presence of obstacles, represented by competing agents.

2.2 Parcels representation

When tracking parcels on the map, the agent separates them into two distinct maps based on their current ownership status:

- `carriedParcels`: contains the set of parcels currently carried by the agent and potentially deliverable;
- `freeParcels`: includes all parcels that, since they were last observed, have not been picked up by other agents, and are therefore considered available for collection;

Both sets are continuously updated by monitoring changes in parcel status through the `onParcelSensing` callback. This mechanism allows the agent to process new information about parcels (for instance, by removing a parcel from `freeParcels` if it is detected being carried by another agent). Additionally, at all times, the agent has access to several environment variables, including `PARCEL_DECADING_INTERVAL`, which defines the frequency at which each parcel's reward decreases. This allows the agent to continuously estimate the current value of all parcels without needing to directly sense them. The parcels' reward is updated at a constant rate, using the function `decayParcels()`: for every parcel, it check if the time passed since the last updates is greater than `PARCEL_DECADING_INTERVAL` and, in that case, decreases its reward in the belief set by the correct amount.

2.3 Agents representation

Within the agent's internal representation, information about other agents is maintained in the `otherAgents` map. For each agent, the system stores and updates its current position on the map, as well as the set of tiles it occupies. Notably, when an agent is in motion, it may occupy two adjacent tiles simultaneously. This information is used to dynamically update the graph representation of the environment: upon sensing the presence of another agent at a specific location, the corresponding tiles are marked as non-traversable. This ensures that the path-finding algorithm accounts for temporary obstacles and avoids planning routes through those tiles.

3 Options generation, filtering and intention revision

There are three main types of intention our agent can adopt during its life cycle, each characterized by a specific identification code:

- `go_pickup`: the agent moves to a given location, in which it believes a parcel is available, and pick it up if it is still there;
- `go_deliver`: the agent moves to the closest reachable delivery tile and, if traversable, puts down the parcels it's currently carrying;
- `idle`: adopted when no other intention is available, encourages the agent to explore the map, focusing on generating cells which haven't been visited for a while.

3.1 Options generation and filtering

Each time new information is sensed, the agent invokes the `generateOptions` function to evaluate potentially valuable actions to adopt as new intentions. This process involves computing the shortest path to the destination of each option using Dijkstra's algorithm.

If at least one reachable option is available (which comprehend picking up parcels or, if the agent is currently carrying any, delivering them) the closest viable option is pushed to the intention set, along with its corresponding path and relevant parameters. If no suitable option is available, an idle intention is selected instead.

3.2 Intention set and intention revision

During its life cycle, the agent repeatedly selects an intention from its intention set and configures itself to pursue it. The intention set is implemented as a priority queue, which is periodically re-sorted based on the utility score of each intention. This ensures that the agent always adopts the intention with the highest current score.

The function `getScore` takes an intention predicate as input and returns a computed utility score, which varies depending on the type of intention:

- `go_pick_up`: the score is computed as `reward - decaySteps - expectedDecay`, where `reward` is the last known reward of the parcel, `decaySteps` represents the estimated reward reduction since the last update, and `expectedDecay` estimates the further decay by the time the agent reaches the parcel;
- `go_deliver`: the score is calculated as `deliveryReward - overallDecay`, where `deliveryReward` is the total reward of all parcels currently carried by the agent, and `overallDecay` is the cumulative expected decay of these parcels before reaching the delivery tile;
- `idle`: this intention always receives a score of `-1`. Since all other scores are non-negative, the idle intention is selected only when no other valid options are available.

Additionally, the agent supports two mechanisms for revising its current intention upon receiving new information. The first method, as previously described, involves re-sorting the intention queue whenever a new intention is added. If the newly inserted intention has a higher score than the currently active one, it is placed at the front of the queue, prompting the agent to abandon its current plan and switch to the new, more valuable intention.Even if the new intention is already present in the queue, it is still inserted and replaces the previous instance. This ensures that the agent maintains an up-to-date version of the intention, including a recalculated shortest path that reflects the latest changes in the environment.

The second method consists in the function `stillValid`, which check if the intention to be adopted is still viable to pursue. Depending on the type of intention, the agent checks the following to determine if it should continue:

- `go_pick_up`: discards the intention if all paths to the parcels are blocked or if it has been picked by someone else;
- `go_deliver`: discards the intention if all paths to the delivery tile are blocked or if our agent is no longer carrying any parcel;
- `idle`: this intention is never really discarded, since pushing a different intention would cause `idle` to stop due to the re-sorting of the intention set.

4 Exploration strategy

When the agent has no immediate intentions to deliver or pick up a parcel, it defaults to an `idle` intention. This fallback behavior is crucial to ensure the agent continues gathering useful environmental information rather than standing still. Over the course of the project, the exploration strategy behind `idle` evolved from a basic proximity-based system to a more refined candidate selection algorithm. Below, we outline this development in detail.

4.1 Initial Idle Movement Strategy

In the first version of the `idle` behavior, the agent followed a simple two-step logic:

- If the agent was *not sensing* any generating cell, it would move toward the **nearest generating cell**, aiming to explore zones where parcels are likely to appear.
- If the agent was already in proximity to generating cells, it would **perform a random walk**, avoiding returning to the tile it had just visited (if possible).

While this heuristic worked reasonably well in smaller or simpler maps, it often led to inefficient exploration in larger or more complex environments. In particular, the agent tended to revisit already explored regions and lacked a global view of optimal exploratory positions.

4.2 Candidate-Based Exploration: The `makeCandidates` Algorithm

To address the shortcomings of the initial strategy, we designed a more principled exploration mechanism, implemented in a procedure we called `makeCandidates`. The core idea was to identify optimal tiles for exploration based on their visibility coverage of generating cells.

The algorithm operates as follows:

1. For each green cell on the map (i.e., generating tiles), calculate how many generating tiles fall within its observation radius.
2. Construct a sorted list of these green cells based on their computed score.
3. Select the top three candidates from this list using the following rules:
   - The first candidate is simply the highest scoring tile.
   - For the second and third candidates, we also factor in the distance from previously selected candidates to ensure spatial diversity.

Initially, the algorithm was computationally expensive, as it checked all generating cells for each green cell. We optimized it by **skipping generating cells that were directly connected** (without any grey tiles in between) to the one currently being processed. This significantly reduced redundant computations and improved performance.

4.3 Improved `idle` Behavior

With the introduction of the `makeCandidates` algorithm, the agent's exploration strategy became more deliberate and structured. The updated `idle` procedure now operates as follows:

- Iterate over the list of candidate positions in descending order of desirability.

- For each candidate:
  - If the tile has **not been visited recently** and a valid path to it exists, update its `lastSeen` timestamp and set it as the current goal using a `go_to` intention.
- If no suitable candidate is found (either due to having no candidates or lack of a viable path), fallback to the original **random walk** behavior.

This hybrid approach allows the agent to explore the map intelligently when good options are available, while still maintaining reactivity and flexibility in edge cases.

Algorithm 1 Select Observation Candidates

```
Require: Set of nodes with positions, Set of generating nodes, Observation distance $d$
Ensure: Set of top candidate nodes
 1: Initialize empty list $Candidates$
 2: Initialize empty set $Processed$
 3: for all generating node $v$ not in $Processed$ do
 4:     Count number of nearby generating nodes within distance $d$
 5:     Identify directly connected generating neighbors
 6:     Add $v$ and its metadata to $Candidates$
 7:     Mark $v$ and its directly connected neighbors as $Processed$
 8: end for
 9: Sort $Candidates$ by number of nearby generating nodes
10: Initialize empty list $Selected$
11: if $Candidates$ not empty then
12:     Add best candidate to $Selected$
13:     if more candidates exist then
14:         Add the one farthest from the first
15:         if more candidates exist then
16:             Add the one farthest from both previous
17:         end if
18:     end if
19: end if
20: return $Selected$
```

5 Multi-agent system

Our multi-agent system builds directly upon the core logic of the single-agent implementation. Agent `A2` reuses the same decision-making, pathfinding, and belief update mechanisms as Agent `A1`, with additional coordination logic for communication and collaboration. This modular design allowed us to extend functionality while maintaining consistent behavior across agents.

5.1 Shared Belief Set and Updated Information

To enable coordination, the agents maintain synchronized belief sets. We extended the belief set by introducing the `otherAgentParcels` field, which stores the list of parcels currently assigned to or carried by the teammate. This field helps avoid conflicts and improves collaborative decision-making.

5.2 Information Exchange Between Agents

Agents communicate exclusively with their designated teammate. Every 100 ms, each agent sends the following updates:

- **Free parcels:** List of currently visible and available parcels.
- **Agent states:** Information about all known agents, including itself (e.g., location, score, path).
- **Carried parcels:** Parcels currently being carried by the sending agent.

In addition to periodic updates, agents send special messages of type `deleteParcel` under these conditions:

1. A known parcel location is now empty (i.e., parcel has expired or been picked up).
2. The agent is picking up the parcel itself.
3. The agent observes another agent carrying the parcel.

5.3 Processing Received Messages

Agents only process messages received from their teammate and ignore those from others.

DeleteParcel messages:

- The parcel is removed from the `freeParcels` list.

Regular update messages: These include three types of data:

1. **Received free parcels:** The agent checks whether the parcel is already known. If not, or if the update is more recent, it adds or updates the entry in its belief set. Parcels already marked as assigned to the other agent are not added to `freeParcels`.
2. **Received agent states:** The agent updates its stored info if the message is newer. It also updates the graph by freeing previously occupied cells and blocking the new ones based on the updated positions.
3. **Received carried parcels:** These are added to the `otherAgentParcels` list to prevent both agents from pursuing the same deliveries.

5.4 Conflict Resolution and Coordination Strategies

Case 1: Both Agents Target the Same Parcel To avoid both agents picking up the same parcel, Agent `A2` always requests approval before picking up a parcel by sending a message to `A1`:

```
Should I drop my intention to pick up parcel ID X? My score for it
is Y.
```

`A2` waits for `A1`'s reply before taking action. The response logic for `A1` is as follows:

- If `A1` has **no intention**, or its current best intention is unrelated to the queried parcel, it replies `"no"`.

- If `A1` intends to pick up the same parcel and its score is higher than `A2`'s, it replies `"yes"`.

If `A1` responds `"yes"`, `A2` drops its intention, adds the parcel to `otherAgentParcels`, and avoids reconsidering it. If `A1` replies `"no"`, `A2` continues with its current plan.

To ensure robustness in dynamic environments (e.g., when `A1` abandons the parcel later due to new blockages or better options), each agent clears the `otherAgentParcels` list every 5 seconds.

Case 2: Parcel Transfer for Delivery If an agent is carrying parcels but cannot reach any delivery point, it attempts to transfer the parcels to its teammate.

This is handled by triggering the new `go_deliver_agent` intention:

- The agent identifies reachable adjacent cells near the teammate (as the exact position is blocked).
- It chooses one such cell as the drop point.
- Before putting parcels down, it checks if it can safely move away afterward using a function called `canMoveAfterPutdown`.
- If movement is possible, the parcels are marked in `otherAgentParcels`, placed on the ground, and the agent moves away.
- It waits briefly to ensure the teammate has time to pick them up.

Table 1: Scores for Single Agent and Multi-Agent Runs (90 secs)

Level	**Agent 1 Score**	**Agent 2 Score**	**Total Score**
Single Agent			
25c1_1	430	-	430
25c1_2	2138	-	2138
25c1_3	691	-	691
25c1_c4	1969	-	1969
25c1_c5	1471	-	1471
25c1_c6	184	-	184
25c1_c7	482	-	482
25c1_c8	930	-	930
25c1_9	1397	-	1397
Multi Agents			
25c2_1	360	382	742
25c2_3	859	1009	1868
25c2_4	709	878	1587
25c2_5	615	480	1095
25c2_6	533	565	1098
25c2_7	1625	872	2497
hallway	0	606	606

6 PDDL integration

Finally, we evaluated the single agent's performance when integrated with a planning mechanism. The planning component was implemented using an API that supports the definition and execution of plans based on the PDDL (Planning Domain Definition Language) standard. Given the initial state, the goal state, and a set of available actions, the API returns a sequence of executable actions that lead the agent toward the specified goal.

Due to the considerable latency introduced by the need to connect to the online API when requesting a plan, the planning integration was limited to the `go_deliver` intention. Extending the integration to include the `go_pick_up` intention as well—combined with the cumulative delays for each planned action—caused the agent to fall behind the continuously decaying parcel rewards. As a result, the agent became stuck in a loop of repeatedly selecting `go_pick_up` intentions.

Given the limited scope of the PDDL integration, map tiles were the only required objects in the PDDL domain. These were defined alongside the `graph` at the beginning of the agent's life cycle. A set of predicates was also introduced to represent the environment:

- `traversable A`: indicates that tile `A` is currently unoccupied and can be traversed by the agent;
- `delivery A`: denotes that tile `A` is a valid delivery location where a `putDown` action can be executed;
- `up A B`, `down A B`, `left A B`, `right A B`: describe the spatial adjacency between tiles, where tile `B` is located in the specified direction relative to tile `A`. These predicates determine which movement actions are available between connected tiles.
- `at A`: checks the agent's current position;
- `canDeliver`: used to verify whether the agent is currently carrying parcels (notice that, given the scope of the planner, there is no need to specify single parcels).

For the actions available to the planner, the following were defined:

- `moveUp A B`, `moveDown A B`, `moveLeft A B` and `moveRight A B`: check if the agent is currently standing in tile `A` and if `B` is traversable and positioned correctly w.r.t. `A`; if so, undeclare the predicate `at A` while declaring `at B`;
- `putDown A`: check if the agent is carrying parcels and is currently standing on `A`, which should be traversable and a delivery tile; if so, put down the parcels and undeclare `canDeliver`.

The problem was submitted to the PDDL planner as a combination of domain objects, the current set of predicates, and a goal state—defined as being located at the destination tile while no longer eligible to deliver any parcels. Although the resulting plan could be executed in full using the provided `PDDLExecutor`, it was decomposed into individual actions. These were fed sequentially to the executor, allowing the agent to continue sensing the environment and revising its intentions between steps if necessary.

While the agent was capable of executing the action sequence returned by the PDDL planner, the latency introduced by the external connection significantly impacted performance. Furthermore, additional safeguards had to be implemented within the agent's control loop to ensure proper synchronization with the planner, adding further complexity to the system.

7 Future Work

While the current system performs reasonably well across various map scenarios, there is significant potential for improvement. Future work could enhance both decision-making quality and computational efficiency through refined scoring strategies, improved coordination, and smarter exploration. Below, we outline several concrete directions for further development.

7.1 Improved Scoring Function

The `getScore` function plays a crucial role in agent decision-making and can benefit from a more nuanced design:

- Introduce a small penalty in the pickup score for parcels that have had multiple nearby agents recently, to reduce agent congestion.
- Consider both pickup and delivery scores together when evaluating parcels, which allows for better comparison and prioritization.
- Replace the fixed score of `-1` for `idleMove` with dynamic scores that account for exploratory or random movements, encouraging smarter fallback behavior.

7.2 Enhanced `idleMove` Behavior

The fallback exploration behavior can be made more adaptive and intelligent:

- Introduce stochasticity into the random movement selection to avoid repetitive or predictable exploration.
- In multi-agent scenarios, implement coordination strategies such as:
  - Dividing the map spatially among agents.
  - Dividing exploration candidates among agents to avoid overlap.
  - Employing real-time coordination to maximize the overall coverage of the walkable area and improve observation of generating (green) cells.

7.3 Smarter `makeCandidates` Algorithm

The candidate generation algorithm could be made more adaptive and informative:

- Allow a flexible number of candidates rather than fixing it to 3; adapt based on map size, agent count, or context.
- Expand and refine the features used to evaluate each candidate:
  - Use the **ratio** of generating cells seen over the total number, rather than raw counts.
  - Consider the **total number of surrounding cells** (including grey/inaccessible ones) to estimate openness or centrality.
  - Factor in **distance to delivery points**, or use the average distance to top-k delivery targets.
- Make candidate scores dynamic:

  - Include **path cost** from the agent's current position instead of just checking reachability.
  - Use the actual **last seen timestamp** of a generating cell or the last time it produced a parcel, instead of only the intention push time.
  - Penalize candidates that are already being observed by nearby agents, to promote diversity in exploration.

7.4 Faster Pathfinding

Pathfinding remains a computational bottleneck, especially with frequent Dijkstra calls:

- Investigate solving memory issues related to Floyd-Warshall or similar all-pairs shortest path algorithms.
- Implement partial caching of frequently used paths to reduce redundant computations.
- Identify and prioritize **critical connector points**—tiles that link otherwise disconnected subgraphs—when computing paths or making decisions.

7.5 Integration of Learning-Based Approaches

Finally, advanced learning methods could greatly enhance agent behavior:

- Use reinforcement learning or neural networks to tune parameters like score weights or exploration thresholds dynamically.
- Train models to learn environment-specific strategies, exploration patterns, or even direct action selection based on observed states.
- Employ learning methods to predict parcel generation hotspots or to adapt coordination strategies in multi-agent settings.