ASA Project Report

FAST: Fast Autonomous & Safe Transport

Daniele Della Pietra and Giovanni Valer*

**The authors contributed equally to this work*

Abstract—In the Single-Agent Scenario, our agent uses a BDI framework for autonomous decision-making, with map analysis and heatmap-guided exploration. For the Team Scenario, we developed a decentralized multi-agent system, adding essential communication for belief sharing and coordination, and using parcel locking and handoffs to avoid conflicts. Additionally, a PDDL-based planner enhances exploration efficiency by guiding agents to visit key spawn tiles. Tested on diverse maps, the system shows strong adaptability, cooperation, and robustness.

Contents

1. Introduction

This report presents our choices and solutions for the challenges of the Deliveroo game. We followed a BDI agent architecture, but deviated from the classical BDI control loop, in that belief revision is performed asynchronously. Having sensing-driven belief updates (instead of revising the beliefs once per deliberation cycle) brings many advantages in this scenario: it improves the reactivity of the agent by reflecting the changing environment accurately, and enhances modularity by completely separating deliberation and belief revision. The code is available in our GitHub repository[1].

We firstly present the architecture we devised for the Single-Agent Scenario, then explain the differences and improvements introduced to tackle the Team Scenario. Finally, we illustrate the usage of PDDL.

In the process of developing our project, we decided to integrate it with the Deliveroo dashboard[2] implemented during the previous academic year by some colleagues of ours. We updated it, in order to make it function with the latest version of Deliveroo, and slightly modified it according to our needs. It really helped to debug and improve our agent, by visualizing its current intentions and beliefs.

2. First Scenario: Single Agent

The map in which the agent is going to play is not known in advance, and can have many different peculiarities. For this reason we argue that computing useful statistics upon receiving the map is highly effective in adapting the agent's behavior accordingly. Such statistics are mainly used for intention selection and revision, but we describe them here as they are calculated in a pre-processing step only once, at the beginning.

2.1. Map Stats

We collect the number of tiles (separated count for each type of tile), and also compute the following boolean parameters:

- **Open field**: whether all (or most) tiles are reachable, i.e., non-blocking tiles.
- **Spawning area**: whether all spawning tiles are in one area of the map.
- **Delivery area**: whether all delivery tiles are in one area of the map.
- **Tunnel**: whether between spawning area and delivery area (if any) there is a tunnel. A tunnel is defined as a minimum number n of consecutive tiles with only two possible moves.

Together with our stats, we also collect important configuration variables from Deliveroo, in order to further adapt both the agent's beliefs and intentions.

Heatmap

Moreover, we define a *heatmap* H of the spawning tiles S, by computing the distance of each $s \in S$ to its nearest delivery tile. We use an exponential function to avoid large differences for huge distances. We want that $h_{x,y} \in [0, 1]$:

$$h_{x,y} = exp\left(-\frac{2 \cdot dist(s_{x,y}, ND(s_{x,y}))}{w_{map} + h_{map}}\right) \quad \forall (x, y) \mid s_{x,y} \in S$$

where:

- $ND(s_{x,y})$ is the nearest delivery tile (for $s_{x,y}$)
- w_{map}, h_{map} are the width and height of the map, respectively

[1] **Code:** https://github.com/jo-valer/fast-agent

[2] https://github.com/lorenzoorsingher/DeliverooDashboard

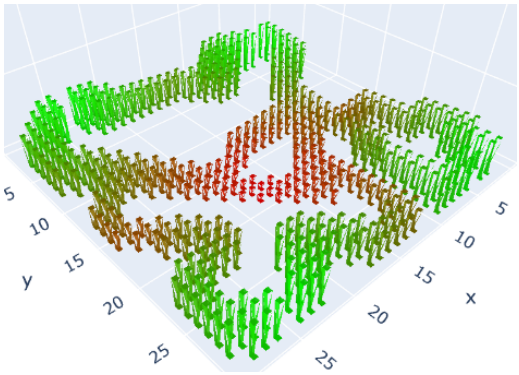

Figure 1. A 3-dimensional visualization of the heatmap.

When $|S| > t$ (with t being a predefined threshold, conveniently set to 200), we initialize with $h = 1 \; \forall h \in H$, to avoid computing the distance between too many pairs of tiles.

The heatmap H is firstly initialized (see Section 2.2) and then continuously updated during belief revision (see Section 2.3.1), and used to select the best area to explore. A value close to 1 means that our agent should prefer that tile, while a value close to 0 means that it should be avoided. A graphical visualization is presented in Figure 1.

2.2. Belief Set

The following paragraphs provide the structure of our belief set, while in Section 2.3 we describe the belief revision mechanism. Our agent knows its own id, position and carried parcels; moreover, it stores also the following information.

Map At first, our agent receives the map from the server and saves it. It also computes the *island* it is in (there are maps that present separated areas, not reachable from one another). This is fundamental in such specific situation, but is also useful in other cases, as the island map does not contain any *blocking* tiles, thus slightly reducing the size and the computational costs. Indeed, we only use the agent's island to compute the statistics and to initialize the heatmap H.

Parcels The perceived parcels have id, position, reward and possibly the carrying agent `id`. We add a new field: **probability**, which has to be interpreted as the *probability of the parcel being there*, thus it is 1 when observed.

Agents For the opponent agents, as in the parcels, we adopt a **probability**. Furthermore, we also add the **last move** field, to track the moves performed by the agents.

2.3. Belief Revision

Our agent continuously senses the environment and updates its beliefs accordingly. Regarding the agent's belief about itself, no particular mechanism is needed, since it constantly receives the correct values.

2.3.1. Map

At every iteration, the heatmap H is updated, decreasing the value of tiles that are close to opponent agents A in the belief set. The weight of each agent is determined by its distance to the tile and its probability. The update of $h_{x,y}$ is the following:

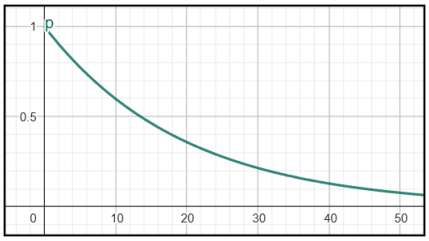

Figure 2. Decaying probability of parcels and agents.

$$h_{x,y}^{(i+1)} = h_0 \cdot (1 - \Delta_{x,y}^{(i)})$$

where:

- h_0 is the value at initialization without agents
- $h_{x,y}^{(i+1)}$ is the value of $h_{x,y}$ at the next iteration
- $\Delta_{x,y}^{(i)}$ is computed according to the formula:

$$\Delta_{x,y}^{(i)} = \sum_{\substack{a \in A \\ \|(x-x_a, y-y_a)\| \leq r}} \alpha \cdot p_a \cdot \exp\left(-\frac{(x-x_a)^2 + (y-y_a)^2}{2\sigma^2}\right)$$

  where:

  - (x_a, y_a) is the position of the opponent agent
  - p_a is its probability
  - σ, r, α are constants

We empirically observed good results by setting $\sigma \in [1, 3]$ and $r = 3\sigma$. Finally, $\alpha = 0.5$ ensures that $h_{x,y} > 0$.

2.3.2. Parcels

When a parcel is sensed, its probability is set to 1 and is added to the belief set. There are two conditions that prevent a parcel to be added: an opponent is carrying it, or an opponent is on its same tile[3]. When a new parcel is added to the belief set, a *new parcel* event is emitted, so that the intention revision loop is aware of the happening. The same conditions stated above trigger the removal of the parcel from the belief set. A parcel is removed also if its reward is 0 (or less), or if its last known position is within the agent's observation distance but the parcel is no more sensed (i.e., we assume an opponent picked it up). When a parcel is no more observed, its reward still get updated according to the *parcel decading interval*. Moreover, its probability decays:

$$p_p^{(i+1)} = \gamma \cdot p_p^{(i)}$$

The best value for γ depends on several different factors, including (but not limited to):

- number of opponent agents,
- agents' observation distance,
- maximum number of parcels,
- parcel's reward,
- size of the map,

[3] If the agent is moving between two tiles, we anticipate its next tile.

- number of spawning tiles,
- agents' movement duration

But we find that just setting $\gamma = 0.95$ (Figure 2) works fine in all scenarios. We initially considered to decrease the probability also in the case opponent agents are observed nearby a parcel, but later discarded this option as it is the planner's job to anticipate opponents actions, so this should not be demanded to the belief revision mechanism.

Algorithm 1 Parcel Belief Revision

Input: *perceivedParcels*

newParcelDetected ← **false**

// Step 1: Handle parcels no longer perceived
for all known parcels not in *perceivedParcels* **do**
  if parcel within observation range or reward=0 **then**
    remove parcel from beliefs
  else
    decrease parcel's existence probability
  end if
  remove parcel from agent's carried list
end for

// Step 2: Incorporate perceived parcels
for all *perceivedParcels* **do**
  if parcel is new and no opponent at parcel's tile **then**
    newParcelDetected ← **true**
  end if
  if parcel is not carried **then**
    if opponent at parcel's tile **then**
      remove parcel from beliefs
    else
      update or add parcel to beliefs
      ensure parcel is not marked as carried
    end if
  else if parcel is carried by our agent **then**
    update or add parcel to beliefs
    add parcel to carried list
  else ▷ *carried by an opponent*
    remove parcel from beliefs
    remove parcel from carried list
  end if
end for

if *newParcelDetected* **then**
  emit "new_parcel" event
end if

Note that the agent keeps low-probability parcels in the beliefs, because we expect the deliberation loop to properly handle such parcels: it is not the role of the belief revision mechanism to decide what options to discard. For better understanding, we provide the pseudocode in Algorithm 1, which also shows the updates to the agent's list of parcels carried by itself. Note that the no-more-observed parcels' reward update is not mentioned in the pseudocode, as it operates autonomously[4].

[4]The decreasing reward of parcels is implemented using `setInterval()`.

2.3.3. Agents

When an agent is observed for the first time, its probability is set to 1, and the last move can be undefined (if observed on a tile) or determined (if observed during movement). If the agent is continuously observed, the last move is calculated looking at its previous and current positions. When an agent is no more observed, its probability rapidly decreases and after few iterations it is removed from the belief set. As with parcels, a *new agent* event is emitted when needed. Note that if the agent is observed during movement, i.e., between two tiles, its position is set to the nearest tile (which is the one it is moving to). This allows our agent to slightly anticipate the environment changes. We experimented with lower values for the decaying factor γ, compared to parcels' probability. This is motivated by the fact that the agents move, therefore their *probability of being there* should reflect this. However, due to the large number of variables, it is difficult to appreciate significant differences. The complete mechanism is provided is Algorithm 2.

Algorithm 2 Agent Belief Revision

Input: *perceivedAgents*

newAgentDetected ← **false**

// Step 1: Incorporate perceived agents
for all *perceivedAgents* **do**
  if agent not previously known **then**
    newAgentDetected ← **true**
  end if
  estimate and update agent's last move
  set agent's existence probability to 1
  update or add agent to beliefs
end for

// Step 2: Handle agents no longer perceived
for all known agents not in *perceivedAgents* **do**
  decrease agent's existence probability
  reset agent's last move
  if agent's probability below threshold **then**
    remove agent from beliefs
  end if
end for

// Step 3: Update heatmap if needed
if map is initialized and at least one agent is known **then**
  update heatmap
end if

if *newAgentDetected* **then**
  emit "new_agent" event
end if

2.4. Desires

In our BDI-based architecture, desires represent abstract motivational states. Essentially, the agent's higher-level objectives that guide intention generation and selection. Our agent maintains the following core desires throughout the game:

- **Maximize Delivered Reward**: The primary objective

is to deliver parcels with the highest possible cumulative reward over time. This desire influences both pickup and delivery decisions and is the main driver of utility computations.
- **Minimize Wasted Movement**: The agent seeks to reduce idle time and inefficient paths. This shapes the desirability of patrolling unexplored areas and influences replanning frequency.
- **Preserve Parcel Opportunities**: Given the probabilistic nature of parcel existence, the agent maintains a desire to act quickly upon newly sensed parcels, balancing opportunistic pickup with ongoing commitments.
- **Monitor and Avoid Opponents**: The agent avoids heavily contested or opponent-dense areas to reduce the risk of losing parcels. This desire is realized via the dynamic adjustment of the heatmap and affects both pathfinding and exploration.
- **Explore High-Yield Regions**: Based on the heatmap and map statistics, the agent develops a desire to explore regions with favorable spawning configurations.

These desires are not mutually exclusive and often compete; the intention revision process (see Section 2.6) evaluates them contextually and translates them into prioritized intentions based on current beliefs and computed utilities.

2.5. Path Search Function

Our path search function is a classic **A* search algorithm** enhanced with dynamic information about currently occupied tiles. The goal is to compute the shortest available path to a given target while considering both static obstacles (e.g., blocking tiles) and dynamic ones (e.g., other agents).

Distance Metric We use the **Manhattan distance** as a heuristic, as movements in the Deliveroo gridworld are limited to four directions (up, down, left, right). The heuristic is both admissible and efficient for grid-based maps.

Blocked Tiles Tiles of type "`0`" are considered blocking and cannot be traversed. Additionally, before the search begins, we **lock** the current positions of all agents. These locked tiles are also considered non-walkable for the duration of the search.

Search Mechanics The function initializes the start tile and uses a priority queue (implemented as a sorted list) to explore neighbor tiles based on the **sum of the current cost (g) and the estimated remaining cost (h)**. At each step, the algorithm:

- Chooses the tile with the lowest estimated total cost.
- Explores its 4 neighbors.
- Skips tiles that are blocking or currently locked.
- Updates neighbors if a better path to them is found.

Plan Extraction Once the goal is reached, the path is reconstructed by traversing the `cameFrom` pointers backward from the goal tile. The output is a sequence of steps, each containing:

- The step number,
- The action taken to reach the current tile (`up`, `down`, `left`, `right`),
- The coordinates of the tile.

Fallback If the goal tile is unreachable (e.g., blocked or surrounded), the function returns `null`. This allows the deliberation loop to react accordingly and possibly trigger replanning or revision of intentions.

Overall, this function provides a robust, agent-aware path planning mechanism that can be used reliably in a highly dynamic environment like Deliveroo.

2.6. Intention Revision

Our intention mechanism is based on the BDI architecture, where goals are expressed as *intentions* and achieved via concrete *plans*. Intention revision includes selecting the best option to pursue, creating new intentions, interrupting current ones when necessary, and executing applicable plans.

Intention Rules

Our agent adopts three top-level intentions:

- **Pick up a parcel**: go to a parcel and collect it.
- **Deliver carried parcels**: find and reach a free delivery tile.
- **Patrol**: actively explore promising spawn areas to spot new parcels.

Intentions are re-evaluated upon sensing relevant changes (e.g., new parcel detected, delivery completed). Each intention type is mapped to a corresponding `Plan` subclass, with priority given in the order: `GoPickUp`, `GoDeliver`, `Patrolling`, and fallback `SearchFunctionMove`.

Options Generation

Upon each re-evaluation, the agent computes the utility of all feasible intentions, guided by the following quantitative models.

Pickup Utility For each parcel p, the utility of picking it up is defined as:

$$U_{\text{pickup}}(p) = \frac{R_p \cdot P_p \cdot e^{-\lambda_p}}{d_p + 1}$$

where:

- R_p is the parcel's current reward.
- $P_p \in [0, 1]$ is the estimated probability of the parcel still being available.
- d_p is the distance (using A*) from the agent to the parcel.
- $\lambda_p = \frac{d_p \cdot t_m}{t_d}$ is a decay factor based on expected time to reach the parcel, where:
  - t_m is the agent's movement duration,
  - t_d is the parcel decaying interval.

The exponential decay $e^{-\lambda_p}$ accounts for the temporal degradation of reward as the agent moves toward the target. Parcels that are likely to be picked up by other agents (i.e., being raced) are assigned a utility of $-\infty$ and discarded.

Delivery Utility When the agent is carrying one or more parcels, it evaluates the utility of delivering them by considering both the potential reward and the time required to complete the delivery. The goal is to prioritize deliveries that are both profitable and timely, while avoiding routes where the reward might decay significantly before the delivery is completed.

First, the agent calculates the total reward R_{total} of all carried parcels, and identifies the parcel with the highest reward $R_{\max}$. It then attempts to find the nearest available delivery tile and determines the shortest path length d_d to that destination.

To account for reward decay, the agent estimates how much time the delivery will take and how that might impact parcels' rewards. The expected number of decay intervals during travel is calculated as:

$$D_{int} = \left\lfloor \frac{d_d \cdot t_{\text{move}} \cdot 3}{t_{\text{decay}}} \right\rfloor$$

where t_{move} is the duration of a single movement step, t_{decay} is the interval at which parcels decay in value, and the factor 3 provides a conservative estimate of decay.

If the estimated decay D_{int} exceeds $R_{\max} + 1$, the delivery is considered unfeasible and is assigned a utility of:

$$U_{\text{deliver}} = -\infty$$

In cases where delivery is feasible, the agent computes the delivery utility using the following formula:

$$U_{\text{deliver}} = \frac{R_{\text{total}} \cdot \ln(1 + R_{\text{total}} \cdot d_d)}{d_d + 1}$$

This function rewards deliveries that provide a high total reward relative to the distance required, while penalizing long or inefficient routes. The logarithmic term ensures diminishing returns for very large reward-distance products, helping to balance risk and reward dynamically.

Fallback Patrol If neither pickup nor delivery yields positive utility, the agent defaults to patrolling. This behavior ensures continuous exploration and system liveness, especially in maps with few or no current parcels.

Options Selection

Generated options are sorted by utility. If multiple options are present, the agent prefers delivery over pickup when utility is higher. Options with low or negative utility are discarded. The top option in the sorted queue becomes the candidate intention.

Intentions Revision

Before pursuing a new job, the current intention is stopped if it differs from the next one. Each job is mapped to a high-level predicate and passed to the `Intention` constructor:

- `["go_deliver"]` for delivery jobs
- `["go_pick_up"]` for parcel pickup
- `["patrolling"]` for patrol
- `["go_to"]` for moving to a location

Intention revision is triggered via event listeners (`new_parcel`, `parcel_pickup`, `parcel_delivery`), ensuring timely adaptation to environmental changes.

Plans Selection and Execution

Each intention loops over available plans in the `planLibrary`. If a plan declares itself applicable (via `isApplicable()`), it is instantiated and executed.

Concrete plan behaviors are:

`GoPickUp:` Sub-intends `["go_to",x,y]` to navigate there, then issues a pickup and emits a `parcel_pickup` event.

`GoDeliver:` Filters for unoccupied delivery tiles, selects the nearest via either Manhattan or A* search (configurable), sub-intends `["go_to",x,y]`, then drops all carried parcels and emits `parcel_delivery`.

`Patrolling:` Uses a heatmap of spawn-tile visit frequencies to pick a high-value target and sub-intend `["go_to",x,y,"patrolling"]`.

`SearchFunctionMove:` (*go_to sub-intention*)

- Computes a path via A* search
- Monitors for racing opponents and can abort on conflict, triggering a replanning.
- Executes each step, performing opportunistic pick-ups and deliveries when passing over parcels or delivery tiles.

A visual summary of the process is shown in Algorithm 3.

Algorithm 3 Intention Revision and Execution Loop

Input: Belief updates and sensing events

```
wait for initial map and tile information
compute reachable spawn and delivery tiles
loop
    if event triggers replan then
        evaluate parcel and delivery options
        sort options by utility and update queue
        stop current intention if needed
    end if
    if queue is empty then
        add patrol intention
    end if
    pop top job from queue
    map job to predicate                ▷ (e.g., go_pick_up)
    create new Intention with predicate
    call achieve(), selecting matching Plan
    if plan fails then
        replan immediately
    end if
end loop
```

3. Second Scenario: Team

To build a team of collaborative agents, we opted for a free-agents approach (no master-slave architecture). Our agents are independent, but communicate in order to actively cooperate and avoid undesirable situations. In the following, we are calling them *buddies*.

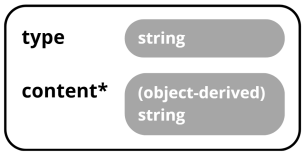

Figure 3. Standard format of messages. Note: *content* is used here as a placeholder; in practice, each message type uses a distinct field name, semantically reflecting its payload.

There are no differences in the map statistics with respect to the Single-Agent Scenario. Note that the heatmap H becomes more powerful in this Second Scenario, since the agent has in its belief set also the opponent agents perceived by the buddy.

3.1. Agents Communication

We exploit the possibility to exchange messages provided by Deliveroo. Both agents know each other `id`, since it is determined by a common configuration file, so they can share with their buddy any useful piece of information. We indeed only use the *say* action: our agents never *shout* nor *aks*, since we want them to be independent and never rely on the other agent. In our approach, explicit communication improves cooperation (and therefore performance), but is not an indispensable requisite, so that an agent is only minimally impacted if the buddy stops communicating. In real-world scenarios, it is quite common for an agent to become unable to communicate due to various factors, such as low battery, poor network connectivity, or other operational limitations.

We decided to exchange only essential information, both in terms of quantity (message length) and in terms of frequency. This was initially motivated by the higher latency caused by messages in Deliveroo, but we later kept this approach as it fits well in our philosophy of having free-agents able to cooperate with an implicit protocol (through observation of the environment) rather than an explicit one (messages), and also because using resources wisely is generally a good idea.

On these grounds, our agents exchange information only about their belief state and intentions (only in specific cases). We have a pre-defined message structure, presented in Figure 3. Note that all objects undergo a conversion to string via concatenation of required fields. To ensure integrity, our agents always verify that received messages were sent by the designated buddy and not by malicious opponents.

3.2. Belief Revision

This section explains the key differences with respect to the Single-Agent Scenario (compare to Section 2.3). There are no relevant differences regarding the agent itself and the map. A key distinction, however, lies in how the buddy's position is updated: each time the agent moves and is outside the buddy's observation range, it sends a message conveying its current position. Section 3.2.2 provides a comprehensive explanation of this mechanism.

3.2.1. Parcels

The logic is very similar to the one described for the Single-Agent Scenario, but with the integration of parcels sensed by the buddy. Parcels' beliefs are updated when our agent:

- observes some parcels
- receives a *parcel* ***perceived*** message from the buddy
- receives a *parcel* ***destroyed*** message from the buddy

In the first case the update is the same as in Algorithm 1, with two additions:

- when a parcel is deleted from our agent belief set, it sends a *parcel destroyed* message to the buddy, transmitting the parcel `id`
- when a new parcel is observed for the first time, a *parcel perceived* message is sent to the buddy, sharing `id`, position and reward

In the second case (upon receiving a *parcel perceived* message), the Step 1 of Algorithm 1 is skipped. In the final case (*parcel destroyed*), the agent simply removes the parcel as instructed by the buddy. A comprehensive overview of the mechanism is provided in Algorithm 4.

Algorithm 4 Parcel Belief Revision (Team Scenario)

Inputs: *perceivedParcels* (list), *fromBuddy* (boolean)

```
1:  if not fromBuddy then
2:      ParcelBeliefRevision(perceivedParcels)
3:      send messages to buddy as needed
4:  else if parcel perceived message then
5:      for all perceivedParcels do
6:          if parcel is not carried then
7:              if opponent at parcel's tile then
8:                  remove parcel from beliefs
9:              else
10:                 update or add parcel to beliefs
11:                 ensure parcel is not marked as carried
12:             end if
13:         else if parcel is carried by our agent then
14:             continue
15:         else
16:             remove parcel from beliefs
17:             remove parcel from carried list
18:         end if
19:     end for
20: else                                  ▷ parcel destroyed message
21:     if parcel is not carried by our agent then
22:         remove parcel from beliefs
23:     end if
24: end if
```

3.2.2. Agents

The agent belief revision differs from the Single-Agent Scenario in that our agents share all observed opponents with the buddy (not only newly seen ones). Moreover, the buddy is never removed from the beliefs, because when it is not observed, a position update message is expected. If this is not received, the probability is decreased, but the buddy never deleted. We do not provide the pseudocode in this case, as it is a straightforward adaptation from the Single-Agent Scenario. Notice only that our agent integrates in the belief set an agent shared by the buddy only if that agent has a probability in the belief set lower than 1 (i.e., it is not observable).

3.3. Intention Revision

The intention revision mechanism in the Team Scenario extends the one described for the Single-Agent case (see Section 2.6), introducing decentralized coordination through lightweight communication. The key goal is to minimize interference and maximize joint efficiency without centralized control.

3.3.1. Coordination Strategy

To prevent agents from pursuing the same parcels and duplicating efforts, we designed a shared-lock protocol inspired by distributed systems. Each agent may temporarily **lock** a parcel when committing to a pickup intention, and announces this lock to the buddy through a message.

The buddy, upon receiving a lock message, tags the parcel as `lockedByBuddy` and excludes it from its deliberation process, treating it as unavailable. This simulates a form of soft mutual exclusion over parcel intentions. The parcel remains locked until:

- the locking agent picks it up,
- the locking agent aborts the plan,
- or a timeout is reached (in case of communication failure or unexpected state change).

This system avoids intention-level conflict and approximates a *Nash equilibrium* of responsibilities: each agent autonomously optimizes its choices while respecting the intentions of the other.

3.3.2. Buddy Drop-off and Handoff

A unique coordination behavior is introduced when path planning fails due to congestion or dynamic obstacles. If an agent carrying parcels is blocked and its buddy is nearby, it may trigger a `stop` message and perform a *handoff maneuver*:

1. The agent puts down all parcels.
2. It moves away from the tile using the inverse of the blocked direction.
3. The buddy observes this event and integrates the newly available parcels into its beliefs.

This behavior enables dynamic parcel handoffs and improves delivery throughput without any global controller.

Why stop the buddy? The buddy may be in the middle of executing a plan, unaware of the new local opportunity. By temporarily halting the buddy's intention execution, the system ensures that parcel handoff is noticed and re-evaluated in the next deliberation cycle. The wait state triggered by `stop` ensures this without requiring deep coupling between the two agents' planning states.

Example Scenario Agent A is blocked and cannot deliver a parcel. Agent B (the buddy) is 2 tiles away and unoccupied. Agent A drops the parcel and emits a `stop` message. Agent B pauses, senses the new parcel, and replans accordingly. This minimizes delivery loss while retaining decentralized autonomy.

An overview of the intention revision mechanism is provided in Algorithm 5. Locking and unlocking are performed using asynchronous messages between agents. If a plan fails (e.g., due to path blockage), the intention is aborted, the parcel lock is removed, and re-deliberation is triggered. This allows the system to adapt reactively to dynamic environments while maintaining decentralized consistency. This protocol provides implicit coordination that is robust to latency and agent failure. It avoids centralization, reduces racing, and ensures a good division of labor with minimal communication.

Algorithm 5 Intention Revision (Team Scenario)

Inputs: *parcels*, *carriedParcels*, *lockedParcels*
Output: Next selected intention

for all parcels p **do**
  if p is not carried **and** $p \notin$ lockedParcels **then**
    $U_p \leftarrow$ computeParcelUtility(p)
  else
    $U_p \leftarrow -\infty$
  end if
end for
$U_d \leftarrow$ computeDeliveryUtility(carriedParcels)
if $\exists p$ such that $U_p > U_d$ **then**
  lock p, notify buddy via message
  return `go_pick_up(p)`
else if $U_d > 0$ **then**
  return `go_deliver`
else
  return `patrolling`
end if

4. PDDL

We hereby present how we integrated PDDL in our project. We experimented with different approaches in order to fully assess the potentiality of PDDL in our scenarios. We tried PDDL for full planning, thus replacing the A* algorithm, but then conveyed that another approach was preferable, as it allows us to overcome a specific problem.

4.1. Patrolling

To enhance agent autonomy and expressiveness, we integrated a PDDL-based symbolic planner to guide the patrolling behavior. This allows the agent to generate tactical exploration routes based on declarative goals, rather than relying solely on reactive heuristics or local search.

Objective

PDDL-based patrolling aims to reach a designated destination tile while visiting as many distinct spawning tiles as possible along the way. This ensures better coverage of high-reward zones and increases the likelihood of encountering new parcels.

Domain Description

We defined a domain called `grid-navigation-patrol`. Locations are modeled as symbolic identifiers (`loc_x_y`), and the agent must navigate from its current location to a target location, passing through key areas. The predicates include:

- `(at ?a ?l)`: the agent is at location `?l`.

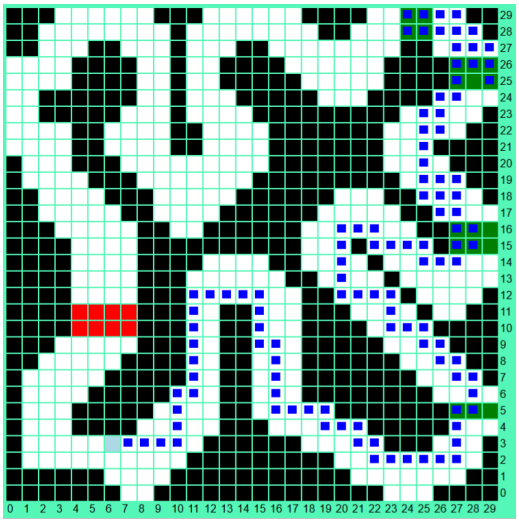

Figure 4. Example of patrolling path found using PDDL. Our agent is on the light blue tile, while the path is marked with dark blue.

- `(adj ?l1 ?l2)`: `?l1` and `?l2` are adjacent.
- `(spawn-loc ?l)`: `?l` is a spawning tile.
- `(visited ?l)`: `?l` has been visited.

Two conditional actions are defined:

- `move-spawn`: transitions the agent onto a new, unvisited spawning tile and marks it as visited.
- `move-any`: transitions the agent onto any tile that is either not a spawning tile or has already been visited.

Problem Generation

When a patrol intention is scheduled, the system builds a dynamic problem instance using the agent's current state. The process includes:

1. Extracting all reachable and walkable tiles.
2. Filtering out tiles occupied by other agents.
3. Identifying spawning tiles and scoring them via the *Heatmap*.
4. Selecting the top half of spawning tiles (by score) as `must-visit` locations.
5. Formulating a PDDL problem where the goal is to:
    - Reach the destination tile.
    - Mark all selected spawning tiles as visited.

Online Solver and Execution

The generated domain and problem definitions are passed to an online PDDL solver. If a valid plan is returned, it is parsed and executed step-by-step as a standard movement sequence. If no plan is found, the agent falls back to A* search.

PDDL Takeaways

This approach demonstrates the integration of symbolic reasoning within an otherwise reactive framework. By using PDDL selectively for high-level exploration, our agents can intelligently patrol their environment without sacrificing responsiveness. It also shows how high-level planning can co-exist with decentralized BDI execution logic, providing robustness and flexibility.

Figure 4 provides an example of PDDL in action: notice how the shortest route is in one direction, but the agent chooses the opposite way to maximize the number of explored spawning tiles. Our solution improves the effectiveness of exploration, enhancing performance by visiting more spawning tiles.

5. Testing

We competed in Challenges 1 and 2, obtaining consistent results in all rounds. These were fundamental real-case validations, that testified the strengths of our solutions, but also highlighted the problems, allowing us to compare different implementations and setups. We collected important insights from the Challenges, and further tested our agents in the provided maps.

Our agents can successfully manage all complexities in existing scenarios. In the following, we provide an overview of some cases we deem important to examine. Each case is accompanied by a list of corresponding maps (if needed).

Non-reachable "islands"

`loops`, `challenge_32`, `challenge_33`, `24c2_8`

This is the case in which our agent can reach only a set of tiles, depending on where it spawns in the map. Such problem is naturally avoided by our agents, thanks to the usage of the *island*, instead of the whole map. It only considers tiles that are on its island for exploration and delivery. Notice, though, that the agent still keeps in the belief set all parcels and agents observed, and shares them with the buddy that might be on a different island.

Narrow corridors

`25c1_5`, `25c1_6`, `challenge_23`, `challenge_24`

The common problem of these maps consists in long and 1-tile-wide passages (but with alternative paths). If these are crossed by another agent in the opposite direction, our agent needs to avoid getting stuck. Through validation in tests and challenges, we proved that a successful strategy is to be *aggressive*, i.e., try to fulfill the current plan until the opponent blocks the way. Then wait (the opponent typically changes direction before we do) and, only if the other agent still blocks us, eventually replan.

A consideration about the Team Scenario should be done; our agents are named a_1 and a_2. When a_1 observes an opponent entering the corridor, this information is propagated to a_2, which will then avoid that corridor when planning. The belief sharing mechanism strongly improves performance in this case.

Forced cooperation

`25c2_hallway`

This is a challenge for the Team Scenario, where active cooperation is mandatory, since one agent has to pick up the parcel, handle it to the team mate, which will then deliver it. This is the only strategy and could be implemented in many

different ways, but we do not need a case-specific procedure, as our mechanism of cooperation (presented in Section 3.3.2) easily handles such situation.

Infinite Observation Range

Our agents do not suffer of performance loss if a high number of parcels and agents are sensed. This is also thanks to the reduced number of messages exchanged by our agents.

6. Conclusion

We have proven the strength of our approach in real-case scenarios through challenges and successfully overcome all issues. Our agents might not be the strongest on each map, but our free-agents approach guarantees high performance in all situations.

Our system demonstrates how a decentralized architecture, enriched with lightweight communication, adaptive heatmaps, and symbolic planning, can handle complex, dynamic environments robustly. The integration of BDI reasoning with PDDL-based patrolling enables agents to balance reactivity and deliberation effectively.

Limitations

The main downside of not having a master-slave approach is that agents, in some particular situations, may step on each other's toes, meaning that they race for the same group of close parcels. Ideally, we would want to have just one agent to pick up a group of close parcels, while the other one does something else (e.g., explore or pick up low-reward parcels). This problem could be tackled by generating a *long-term* plan that considers not only the next best action, but a sequence of actions. However, due to Deliveroo's rapidly changing environment, long-term plans may quickly become obsolete, requiring frequent replanning, making such improvement negligible. Note, indeed, that our mechanism of "booking" parcels avoids this problem in the vast majority of the cases.