Autonomous Software Agent

Razvan Stefan Manole[b,2] **and Emanuele Poiana**[a,2]

[a]*emanuele.poiana@studenti.unitn.it*
[b]*razvanstefan.manole@studenti.unitn.it*

professor Giorgini

Abstract—This is the project for the Autonomous Software Agents course 2025, it consists in different implementations of autonomous software agents operating in a delivery based game environment. Agents observe their surroundings, plan optimal routes to pick up parcels from dynamically spawning locations, and transport them to designated delivery zones. We develop both single-agent and cooperative multi-agent solutions, together with a PDDL extension of the single-agent. Each agent maximizes its scores through utility-driven intention selection, optimizing the path to follow, assigning priorities to different parcels t pick up and avoiding other agents. The multi agent software in addition, performs a handshake protocol to exchange identities and continuously shares beliefs about parcel availability and each other's positions, enabling workload division and collaboration under certain situations. Our architecture followed a Belief–Desire–Intention (BDI) loop, adapted and built ad hoc for our solutions.

Contents

1. Introduction

In this project we focused on building an autonomous agent that will go to pickup the package and will deliver it to a delivery zone. The project is divided into three main parts:

- **single agent:** performs a BDI loop which controls its belief, permits to take actions based on those, like move towards a target destination, pick up and deliver parcels, while avoiding obstacoles like other agents and walls.
- **PDDL agent:** is an extension of the single agent which in hard situations, switches the single agent order of actions to a planned one through a PDDL planner.
- **multi agent:** two agents, based on the single agents communicate and act to cover different regions of the map to maximize their joint score, in some situations they can collaborate communicating a series of messages and by doing specific actions to pass parcels to the companion and get more points.

1.1. Motivation

The primary goal of this project was to implement a fully autonomous software agent capable of exploring a dynamic environment, opportunistically collecting parcels from spawning tailes, and delivering them to designated drop-off zones. Building this system deepened our understanding of the BDI (Belief–Desire–Intention) architecture by guiding an agent through perception, deliberation, and action phases in real time. Moreover, we extended the framework to a cooperative multi-agent scenario: agents communicate, hand off tasks when one fails, and even integrate a simple PDDL-based planner to govern complex delivery sequences. This dual single- and multi-agent implementation not only validated core BDI concepts but also demonstrated the benefits and challenges of distributed coordination in a rapidly changing environment.

1.2. Objectives

The objectives of this project are:

- Design and implement a BDI-based agent: Build a single autonomous software agent that perceives its environment, formulates goals, and executes plans based on the information available. This practically picking up and deliver parcels in the most efficent and generalizable way possible
- Extend to cooperative multi-agent operation: Enable a team of collaborative agents to better solve

the problem sharing information about parcel locations, bottleneck (tunnels) and map coverage.
- Incorporate planning via PDDL: Demonstrate how a symbolic planner can be plugged into the BDI cycle to generate higher-level delivery sequences, dynamically adapting to newly spawned parcels and changing map conditions.

2. Problem Statement

Agents move in a 2D discrete tile map, where are present four types of tiles:

- *Wall*: inaccessible tile (type 0)
- *Spawn*: parcel spawning tile (type 1)
- *Delivery*: parcel delivery tile (type 2)
- *Path*: accessible tile (type 3)

Any parcel can be delivered in any delivery tile, multiple parcels can be carried by a single agent simultaneously. When the game starts the agents know the map in its entirety; have the position for spawning tiles delivery tiles, wall tiles and path tiles. They have partial observability defined at the beginning of the game as a range of visible agents and parcels, the two ranges values are separated. Parcels and agents information within the range are sent at every server tick. When a parcel is delivered, the agent score increases as much as the value in that istant of the parcel. The main objective for each agent (or team) is to maximize the score.

3. System Architecture

3.1. High-level Architecture

The Belief-Desire-Intention (BDI) is a classical implementation when creating an agent. The following schema is a high level explanation of how the project works in order to understand the behavior of the agent and its revision.

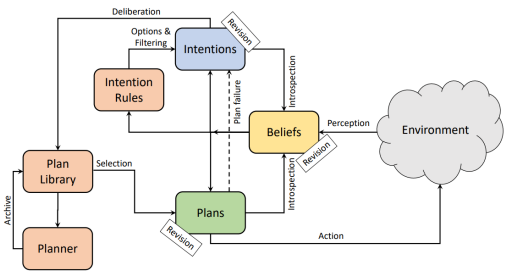

Figure 1. the BDI control loop

- **Belief:** The environment will send to the agent at the beggining of the map the settings, the agent will save those parameters in the belief-set.
- **Desire:** here we have the goals, after getting the belief-set will create its goal to achive
- **Intention:** The intentions are the actions to achive our desire like pickup, delivery and wandering.

The following Flow diagram illustrate our high level implementation of the agent (Fig. 2).

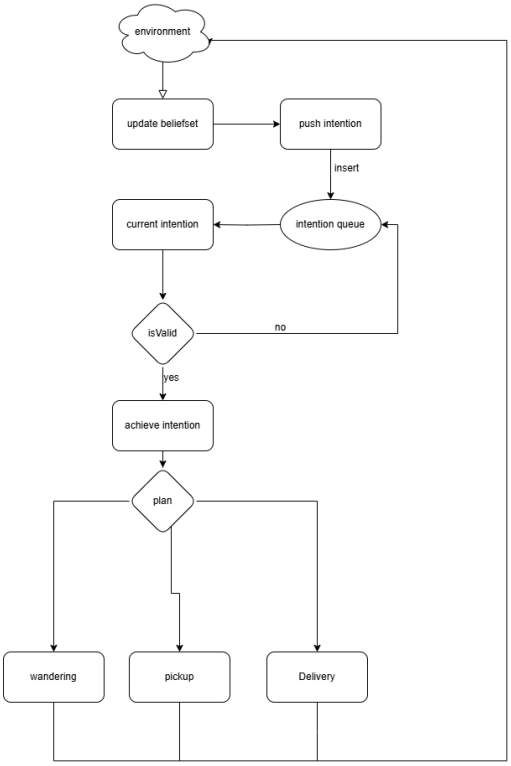

Figure 2. Flow diagram schema

3.2. Code Structure

We have divided the code into classes in order to create a cleaner code and more understandable. With this kind of approach we were able to collaborate and take different tasks without interfering with each other (See Figure 3).

3.2.1. Agents

- **lib**: In the lib directory we have the algorithms.js file where we include all the algorithms used in the project such as BFS. It also includes functions in order to make the code cleaner and not repet same parts of code.
- **main_*.js**: The main folder contains all the entry point files used to launch the agents and start the simulation on the connected map. We have chosen to separate the three agent types: single agent, PDDL agent, and multi-agent. To run the single agent on a local server, you can use main_local.js. To run it on a remote server, use main_server.js. For the PDDL agent, a local PDDL server must be running. Once active, you can launch the PDDL agent on the local map using hybrid_pddl_main.js. If the client wishes to run it on a remote server, they simply need to update the URL specified in the file.

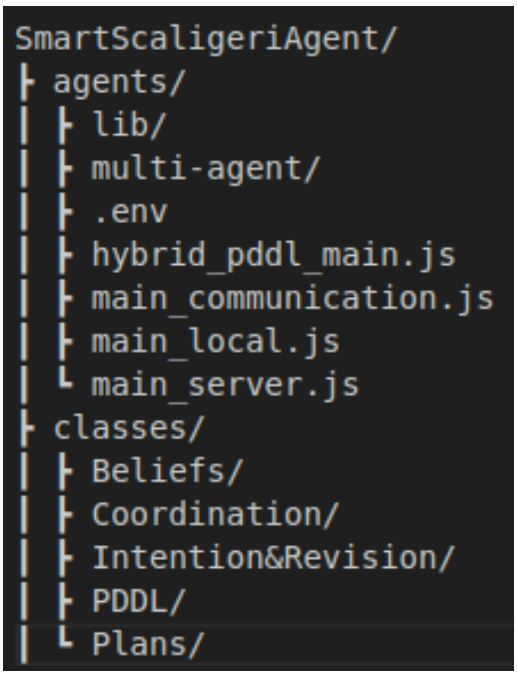

Figure 3. Directory code structure

3.2.2. Classes

- **Beliefs**: The Beliefs directory contains all the agent's beliefs as well as the map settings. If the scenario involves a multi-agent setup, it also includes the beliefs of the companion agent. From the Beliefs, we can extract all the map configurations and the specific settings related to our agent. In the case of a multi-agent setup, the companion's settings can also be retrieved. Additionally, the Beliefs allow us to filter the original map to identify the destination tiles and spawning tiles.
- **Coordination**: The Coordination directory contains the Communication.js file, which handles all message-based communication between the two agents. This file includes the function responsible for the initial handshake at the start of the run, allowing the agents to exchange and update information about each other. It also manages the special case where only one agent is able to deliver parcels, while the other can only pick them up.
- **Intention&Revision** The Intention&Revision directory contains the core logic of the agent, It includes the following files: intention.js, IntentionRevisionRevise.js, and MultiIntentionRevisionRevise.js. It coontains the intention loop, the check of the validity of the intention with isValid and the method achive wich will try to achieve the intention. In IntentionRevisionRevise.js, the push method is implemented to prioritize and push the intention with the highest reward, calculated using a reward function described in the Strategy section4.1. This file also handles the validation of intentions and replaces the current one if a better alternative with a higher reward is found.
  The MultiIntentionRevisionRevise.js file extends IntentionRevisionRevise.js and includes additional logic for multi-agent scenarios. This class contains methods to manage the division of the spawning map between the two agents4.3.2 and handles cases where only one agent can pick up parcels while the other is responsible for delivering them, using a request-based coordination mechanism.
- **PDDL**: The PDDL directory contains the PddlIntentionRevision.js file, which defines the belief set used by the PDDL agent to generate plans. It also includes the PddlIntentionRevision class, which handles the replanning process—this can only occur after a specified replan_time.
  Additionally, the directory includes the fluent-problem.pddl file, which describes the problem definition for plan generation, and the fluent-domain.pddl file, which defines the domain and available actions for the PDDL planner.
- **Plans**: The Plans directory contains the plans.js file, which defines the general plans available to all agents. These include behaviors such as BFSMove, PickUp, PutDown, Idle, and WanderingFurthest. Each specific agent type has its own additional plans that extend the general set defined here. The file also includes subPlans, which allow a plan to be broken down into smaller and more manageable components.

4. Agent Implementation

For the implementation of the agent there have been created 3 kind of agents: the single agent, the multi agent and the PDDL agent. The single agent is considered as the "baseline" implementation; in fact the PDDL and the multi-agent version are extensions of this model. All three autonomous agents have the same high-level set of desires: **maximize parcels delivery in a given time**. This translates to different **intentions**, chosen based on the given **beliefs**.

4.1. Single Agent

4.1.1. Overview

The single agent represents the core of the project, serving as the foundational element upon which all implementation and extensions are built. The agent follows an adaptation of the BDI architecture.

1. In the **Belief** phase, the agent perceives and stores essential information from the environment within its belief-set. These stored data include the initial map structure, spawning locations, delivery zones,

sensed parcels , and the agent's own information, such as its current position, id, name, and score. The agent continuously updates its internal state as it navigates and interacts with the environment.
2. The **Desire** phase is implicitly represented by the intention queue, which maintains a prioritize list of specific goals the agent aims to achieve.
3. In the **Intention** phase, each intention is taken from an intention queue and then concretely translated into executable plans. These plans invoke sub-plans to handle more detailed actions in order to fulfill each intention. New intentions are dynamically generated based on environmental perceptions, particularly when new parcels appear. Intentions can be classified into three types: pick up, delivery, wandering.

4.1.2. Strategy

The strategy we've adopted for the single agent revolves around a cycle consisting of perception, belief updating, intention formation, validation and execution.

Belief updating: every time a new parcel is sensed a pick up intentions for that parcel is pushed to the intention queue other than added to the belief set. Also when an agent is perceived the belief set is updated. Based on its belief, the agent keeps an alternative version of the original map, accessible tiles, which contains only the tiles accessible and not occupied by other agents.

Agent loop: the agent loop is the main component that represent the implementation of the BDI loop (Code 1).

```
loop () {
 while ( true ) {
  if not Idle {
   if intention_queue not empty{
    current_intention = intention_queue.pop
    if isValid(current_intention)
     achieve current_intention
   }
   else wandering
  }
 }
}
```

Code 1. pseudocode of the single agent main loop

At the beginning the agent in the main loop will instantiate the belief-set of the agent took from the environment. Then the agent will enter in the intention loop where will take the first intention from the intention queue to check if the intention is still valid using the method isValid (see Code 3). If the intention is still valid then the agent will try to *achieve* it, where will search the right plan among all the plans from its plan library. The agent will call for the execution of the plan, if the plan finishes successfully the the intention is considered achieved. *If the intention queue becomes empty*, the agent will execute by default a *wandering* plan, exploring increasingly distant spawning tiles in search of new parcels.

Intentions handling: the agent senses its environment and when detects new parcels, it immediately updates its internal belief-set and generate corresponding pick up intentions. These intentions are inserted into an intention queue based on a prioritization determined by the following **reward function**:

$$f_{reward} = \frac{1}{|distance| \cdot \frac{1000}{movement}} \cdot 1000 \qquad (1)$$

this reward function takes in account the movement time of the agent and the distance from the agent to the parcel, the furthest is the parcel the lower is the reward. The result will be then added to the actual reward of the parcel.

$$final\ reward = f_{reward} + parcel_{reward} \qquad (2)$$

After the pick up intention has been added to the queue, a delivery intention tries to be added too. The delivery intention has a lower reward of the parcel just added, this creates implicitly a *"bucket"* intention queue.

```
intention_queue = [pick_up2:15, pick_up1:14,
    ↪ delivery: 12, pick_up3: 8, ...,
    ↪ wandering: 0]
```

Code 2. Intention queue *"bucket"* representation

The reward function (Eq. 2) is not only used to select the best parcel to pick up, but also for order the intention queue. Wandering and delivery intentions are handled by this function, which assigns 0 score to wandering intentions, following the idea of look for new parcels only when I delivered all the ones that I scheduled to pick up and deliver. Every time an agent selects an intention to execute through its loop (see Code 1) from the intention queue, will check through the method *isValid* (Code 3) to calculate if the parcel is reachable, if has been taken from another agent or if it has expired.

Once validate, the agent executes the associated plan, dynamically the problems that can arise during the execution. Issues such as unavailable paths, parcel expiration or taken from other agents are detected and the current intention will be deleted, enabling the agent to reassess its priorities and select a new intention. The Plans represents the sequence of actions to take to achieve an intention or part of it, in fact some plans are divided into multiple sub-plans. Finally, after completing each action, the agent updates its belief to keep the most recent state of the environment.

```

isValid(intention){

reachable = true
    if intention is go pick up
        actual_steps = BFS(me, parcel) +
    ↪ distance_nearest_delivery_tile

        if exist decay step
            if parcel_decay * steps_number <
    ↪ actual_steps)
                reachable = false

        if parcel is not already token and is
    ↪  reachable and still exists
            return true
        return false
}
```

Code 3. pseudocode isValid

4.1.3. Discussion

Overall we addressed the main functionality that are expected from a single agent, following a brief comparison between the advantages and limitations of our agent:

- **Advantages**: The bucket intention queue structure derived from the reward function and the intentions checks before executions, ensures that our agent take reliably some of the most valuable parcels while ensuring it will deliver them, not wasting actions on deprecated plans. The usage of class level shared attributes permits to have updated belief and so variables(like map accessibility or agents positions) to use, avoiding inconsistent actions. The plans encapsulate actions to do, and have fault control logic to ensure alternative actions in case of unusual situations not predictable when generating the parent intention. Furthermore, plans modularity grants easier creation of complex intentions.
- **Limitations**: In some situations, where many parcels are present and the spawn frequency is high, our implementation might result un-optimal and spend too much time picking up parcels without actually deliver them. Other agents doesn't interfere with our pathing due to the removal of their position from the accessible tiles, but no inference of their intentions is made and utilize to improve our intentions generation and schedule.

4.2. PDDL Agent

4.2.1. Overview

PDDL agent is an extension of the 4.1 single agent, when the parcels *pick up* intentions scheduled reach a threshold value, it triggers a planner that based on the parcels position and their relative distance to each delivery tile returns an optimized plan. The plan is in the form of a sequence of parcels to pick up and deliver in specific delivery tiles. If a plan is found, is decomposed in its corresponded (single agent) intentions and executed. When the plan execution is finished it returns to the baseline single agent loop.

4.2.2. Strategy

Given the single agent baseline, which uses plans and sub-plans to achieve a determined intention with mechanisms to prevent collisions with other agents, picking up or putting down parcels in undesired situations(like trying to pick up a parcel in a empty tile), we decided to use those function as the executive part of each step found by the PDDL planner. The main features of our approach are:

```
(define (problem <problem-name>)
  (:domain <domain-name>)
  ;; Declare all objects, grouped by
    ↪ type
  (:objects
    agent1 - agent ;agents definition
    loc1 loc2 loc3 loc4 - location ;
    ↪ objects locations (agent, parcels,
    ↪  delivery tiles)
    parcel1 parcel2 - parcel
    tile1 - delivery_tile
  )
  ;; Initial state assertions
  (:init
    (at agent1 loc1)
    (at-parcel parcel1 loc2)
    (at-tile tile1 loc3)
    (= (cost-of-move loc1 loc2) 1)  ;
    ↪ Example cost
    (= (cost-of-move loc2 loc1) 1)  ;
    ↪ Example cost
    ...
    (= (total-cost) 0) ; initialize the
    ↪ total cost that have to be lower
    ↪ than "threshold" ;variable
    (= (threshold) 10) ; threshold
    ↪ variable
  )
  (:goal
    (and (delivered parcel1)
        (delivered parcel2)
        ... ; all the other parcels
    )
  )
)
```

Code 4. template PDDL problem definition

PDDL Domain and Problem definition: we chose to invoke the Planner for finding the optimal sequence

of parcels collection and delivery. To do so the domain has been defined as a list of three actions, move, pick up and deliver. The problem defines all the variables that are a graph like declaration of all the distances between all the parcels and the delivery tiles. See Code 4 for more details on the problem definition.

PDDL-loop: this extension of the single agent loop enables the PDDL agent to call the planner given its actual belief set and intention queue to find a plan which pick ups all the parcels scheduled for by the respective intentions in the queue. This is controlled by an hyper-parameter N that is compared with the number of pick up intentions in the queue. If it is greater, then the single agent loop and push methods are suspended, and starts the planning procedure. The algorithm initialize the PDDL problem with the belief set variables, then tries to solve the problem, if it finds a plan, it decomposed in its fundamentals parts *move*, *pick up* and *deliver* that are translated into the equivalent intentions and so plans and sub-plans and executed. When all the intentions are reached or in case of failure, the PDDL loop ends and the agent returns to the baseline loop. See Code 5 for more details.

```
PDDL-loop (N: threshold number, i_q:
    ↪ intention queue) {
 while ( true ) {
  if(i_am_idle){
   // count the number of parcels pick
    ↪  up intentions
   pick_up_number = i_q.count("pick_up
    ↪ ")
   if ( pick_up_number >= N){
    //change the Pddl flag to
    ↪ deactivate the standard push
    this.#Pddl = true;
    //call the PDDL planner and
    ↪ execute the plan
    await getAndExecutePlan();
    //go back to single agent
    this.#Pddl = false;
   }
   else{
    //continue with the single agent
    ↪ loop
    single_agent_loop();
   }
  }
 }
}
```

Code 5. pseudocode PDDL-loop

Replan: during the execution of any PDDL plan, is frequent that the agent perceives new better parcels, so we added the possibility to re-plan the agent actions. This comes with two challenges, how and when to re-plan. Addressing the first one, it was straight forward just stop the current plan, recall the PDDL planner and execute the new (if found) plan which yields an updated belief-set. It comes clear that the action of re-planning is expensive from a time perspective, so we decide to:

1. Switch to the PDDL routine when a new perceived parcel reward function is greater than the one of the current target parcel. This avoids to re-plan for low value parcels
2. Re-planning more times in a row is possible only every 30 seconds, to avoid getting stuck re-planning indefinitely.

Optimization score The PDDL planner returns the first valid plan it founds, which frequently is not the best but in many cases is among the worst. So we defined a **threshold** that the PDDL has to respect to return a valid solution, this is done by making each moving action tied to a cost that increases the *total_cost* variable in the problem definition.
The threshold is computed as follows:

Given ($p \epsilon P$) set of all the given parcels, a agent, $d \epsilon D$ set of all the delivery tiles and $d(x, y)$ Manhattan distance between two 2D points

$$\text{AgentAvg} = \frac{1}{|P|} \sum_{p \in P} d(a, p) \tag{3}$$

$$\text{ParcelAvg} = \frac{2}{|P|(|P|-1)} \sum_{\substack{p_1, p_2 \in P \\ p_1 \neq p_2}} d(p_1, p_2) \tag{4}$$

$$\text{DeliveryAvg} = \frac{1}{|P|} \sum_{p \in P} d(p, d_p) \tag{5}$$

$$\text{Threshold} = \text{AgentAvg} + \text{ParcelAvg} + \text{DeliveryAvg} \tag{6}$$

This guarantees not the optimal solution but at least that the returned one has some form of optimized path which is shorter than the average one.

4.2.3. Discussion

- **Advantages**:
  The agent extends the single agent, in this way if the agent doesn't find a plan will start with the single agent algorithm. We are sure that the agent will always deliver because will put always a delivery at the end of the plan and will not be blocked if will see another parcel near by. The threshold6 helps in finding a better plan.
- **Limitations**: When the agent will change the plan will be stacked for a few moments and in that time the adversarial environment could change, and if the agent doesn't find a good plan it has wasted the time in searching it. The agent performs bad against

othe players because the plan doesn't predict if the parcel will be still in the previous position.

4.3. Multi Agent

4.3.1. Overview

For the multi agent part of the project the we have two agents that are in the same team, but collaborate only in special cases, acting as simple single agents 4.1 in the majority of their activity.

4.3.2. Strategy

The strategy adopted can be seen as two separate functionality which modify the behavior of the single agent baseline. One is right at the beginning of each agents execution, where the two agents coordinates in a master slave schema with fixed roles to divide the map they are patrolling in a equal way. The other is situational, and it's related to the possibility for either agent to deliver parcels(that is carrying).

Map division: to implement a map division for the team agents, we decide to divide only the tiles visited during the *wandering* phase and the delivery tiles in this way each agent can go in all the map's accessible tiles but focuses on only the assigned spawn and delivery tiles areas. The agent with the master role computes the two categories of tiles accessible for him at the beginning of the execution, then keeps the half nearest as its wandering and delivery tiles set and finally sends the remaining to the friend agent. If the map has many spawning tiles, we switched as distance metric from the minimum actual distance(minimum number of tiles to go from one tile to another arbitrary accessible one) which is expensive to compute because we are using the BFS, to the Manhattan distance between the agent and every spawning tile. See Figure 4 for more details.

Parcel exchange: in some maps, especially the "hallway" might happen that one of the two agents(or both) is incapacitated to deliver or pick up parcels. This is caused by other agents or for the map morphology, which has "islands" regions that are not connected to spawning or delivery tiles. Excluding the second case, not present in this project, we address the first case by dividing the problem in 2 sub-phases when encountered.

1. *Obstacle detection*: when one of the two team agents can't reach any delivery tile but it's the teammate closing the passage in the map, it asks (the one which has the parcels and can't deliver) for a parcel exchange. See code 6 for more details.
2. *Parcel exchange procedure*: when the exchange procedure is invoked, the two agents coordinates in a Master Slave schema, where the master is the one that can't deliver its parcels and will start the communications with the other allied agent. During this phase, when the slave agent receives the meet request, it firstly stops all its actions and intentions, then after the slave acknowledgment response, the master move to its position, drop the parcels, and move away. After this the master will send a request to the slave to pick up the parcels and waits until it reply after has picked up the parcels. Then the slave proceed to deliver the parcels while the master re-establish the single agent BDI loop, same thing done by the slave after it has delivered the parcels. See figure 5 for more details.

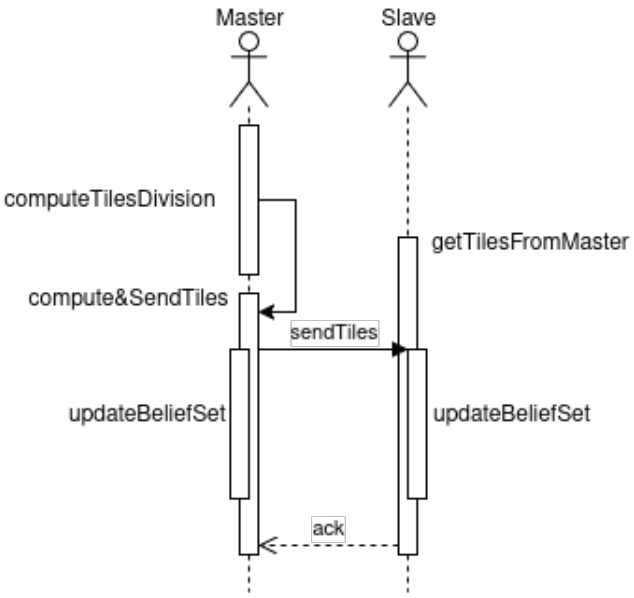

Figure 4. Map division temporal diagram

```
delivery(me,delivery_tile,og_map){
 //Taken from the updated beliefSet
 accessible_tiles,friend_position
 path = BFS(me,delivery_tile,accessible_tiles)
 og_path= BFS(me,delivery_tile,og_map)

 if path not found but og_path is found
  check if the friend_position is in the
    ↪ og_path
   start exchange procedure
 else if during the move actions, a tile gets
    ↪  occupied
  check if the friend_position is in the
    ↪ occupied tile
   start exchange procedure
  else single agent delivery
 else single agent delivery
}
```

Code 6. pseudocode obstacle detection implemented in delivery

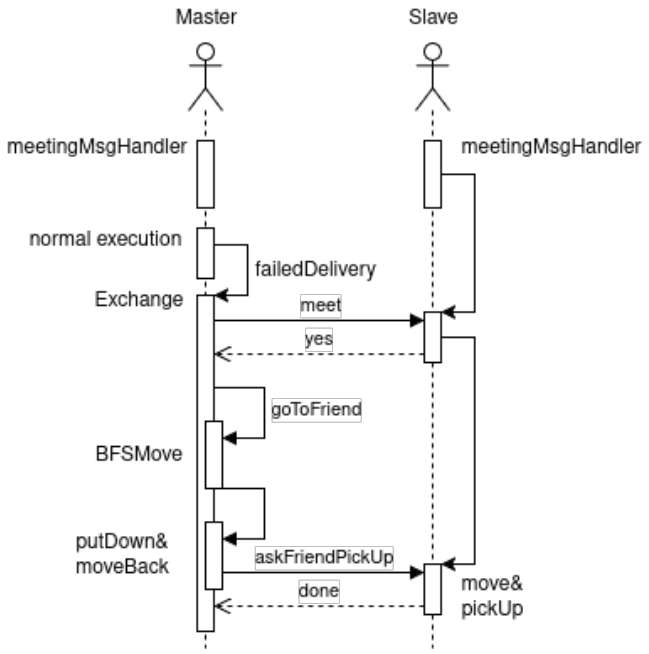

Figure 5. Parcel exchange temporal diagram

4.3.3. Discussion

- **Advantages**:The two agents operate independently from each other, allowing them to navigate the map more efficiently. If an agent is able to pick up and deliver a parcel, it will do so autonomously, without waiting for coordination, this ensures that the process is not slowed down. If one agent is looking for parcels to pick up and perceives any parcel on the other's portion of the map can go there to collect it and this is optimal when the teammate agent is busy in other intentions. The map is divided only once at the beginning of the game to minimize communication, further promoting agent independence and reducing unnecessary interaction.
- **Limitations**: The two agents can interfere with each other's designated area of the map. Since there is no exchange of information about the intentions between the two agents for parcel collection, in fact if one agent sees a parcel in the other's territory, it will go to pick it up, even if it was already in the companion intention to pick it up. Because they operate independently, they have no mechanism, other than the parcel exchange, to collaborate; so even if their own half is empty, agents are still coded to go in the same area of the map. Oppositely, even if their area is valuable, they can go anyway in the friend's map zone, reducing the total achievable score. Another problem which arise is the fact that agents exchanges parcels only when there are no other options, which means that can interfere with each other path and take longer path instead of just passing the parcels. One of the main issues with this setup is that each agent constantly tracks the exact movements of its companion, which introduces delays and can make the agent slower.

5. Results

All three agent implementations run successfully on all the given maps, but clearly some maps are more suitable rather than others for each agent version. We set a baseline measure in every map, which represents the average time to take a parcel and deliver it from a naive agent implementation, one that pick a parcel and delivers it and do no more considerations about parcels priority, multiple picks before delivery and so on. The target score is computed as followed:

$$\text{AvgDistance} = \frac{\sum_{i=1}^{m}\sum_{j=1}^{n} d_{ij}}{m \times n} \tag{7}$$

$$\text{Target} = \frac{60}{(\text{AvgDistance}) * 2 * t} \tag{8}$$

With avg. time to do one step in each map.

All the experiments have been runned five times for 1 minutes each and the reported score is the average of these runs. The single agent delivery reward value was set to half of the parcel reward value. For the PDDL N was set to 3.

Map	Single	PDDL	Target	Ratio
25c1_1	270	60	60	**4.500** / 1.000
25c1_2	934	37	300	3.113 / 0.123
25c1_3	500	522	240	2.083 / **2.175**
25c1_4	1050	1100	700	1.500 / 1.571
25c1_5	1070	487	400	2.675 / 1.217
25c1_6	268	255	330	**0.809** / 0.773
25c1_7	471	247	240	1.963 / 1.029
25c1_8	366	27	270	1.356 / **0.100**
25c1_9	1838	821	520	3.537 / 1.578

Table 1. Single and PDDL agent comparison

Map	Multi	Target	Ratio
25c2_1	658	480	1.371
25c2_2	406	360	1.128
25c2_3	1068	800	1.335
25c2_4	1530	520	**2.942**
25c2_5	475	480	0.989
25c2_6	1116	700	1.594
25c2_7	864	700	1.234
hallway	369	300	**1.230**

Table 2. Multi agent performance

5.1. Discussion

The results confirmed most of our thoughts and hypothesis on the strong a weak points of our agents.

5.1.1. Single agent vs. PDDL agent

Single agent implementation clearly dominated the PDDL version in terms of score maximization, but this is mainly because the PDDL planner is quite slow even if hosted locally compared to the speed at which all the logic operation are performed, such as BFS, array sorting, comparison; or the speed of the environment, often generating many new parcels while the PDDL planner is still finding a solution to an outdated belief set. The single agent tends to stay in areas with frequent spawning parcels, delaying considerably the delivery, while the PDDL ensures the delivery of the selected parcels, even after a re-planning thanks to the time constraint between re-planning operations. This could become a stall situation for the single agent where, even if it has one or more parcels, doesn't deliver.

5.1.2. Multi agent

Overall the multi agent performs almost always above the target score, in the hallway the parcels exchange works and in fact we obtain a higher score than the target one (computed referencing one agent because two have to cooperate). The lack of coordination causes to interfere each other path in some narrow shaped maps, causing both agents to re plan their path and choose worse paths.

6. Conclusions

The agents work and perform as expected; this is shown in the results where we often surpass the target score. During the development the code structure changed over time, switching to a class based project, making easier future improvements and working in parallel on different functionalities. This modularity was crucial to our approach of extending the single agent baseline.

6.1. Future Works

We encountered many complex situations to handle and some solutions were not implemented due to the time needed to add them compared with what we considered to be little improvements to the actual performance. Here are some of the most interesting we haven't include:

- *The path-finding algorithm* of the agent can be replaced by other algorithms which could be more optimal in finding a path to the delivery and spawning tile.(like A*)
- *The reward function* could also consider the probability of being able to pick up a parcel focusing on the adversarial side of the project. Also have a dynamic adjustment of the delivery intention considering the elapsed time from its add to the queue or the number of carried parcels could be beneficial.
- *The wandering algorithm* could be improved with a heatmap where the agent will try to search for new parcels in zones where there are less players in order to have more chances to find new parcels.
- *In the multi agent communication* communicating intentions in order to not pick up a parcel if the other one had already it in its intention queue maximizing the potential score.

7. Reference

1. Ai-Planning. (s.f.). GitHub - AI-Planning/planning-as-a-service: The newly improved planner (and more) in the cloud. GitHub. https://github.com/AI-Planning/planning-as-a-service
2. Unitn-Asa. (s.f.). GitHub - unitn-ASA/Deliveroo.js: Multiplayer Grid-based Parcel Collection Game for Educational Purposes in Planning and Autonomous Agents. GitHub. https://github.com/unitn-ASA/Deliveroo.js
3. Unitn-Asa. (s.f.-a). GitHub - unitn-ASA/DeliverooAgent.js. GitHub. https://github.com/unitn-ASA/DeliverooAgent.js
4. IlPoiana. (s. f.). GitHub - IlPoiana/SmartScaligeriAgent: Repository for the 2025 AutonomusSoftwareAgents course project. GitHub. https://github.com/IlPoiana/SmartScaligeriAgent?tab=readov-file