Single and multi autonomous agents approaches in a “parcels delivering” web-based game

Davide Modolo (#229297), Davide Moletta (#239272)

Abstract
The goal of this project was to develop an autonomous software agent able to play the web game Deliveroo.js [1]. The agent follows a BDI architecture to sense the environment and manage its beliefs and intentions. The objective of the agent is to maximize the points that it can earn by picking up and delivering parcels in the game. In this report we will explain the logic behind our agents, how we implemented the BDI architecture and all the explored approaches (single-agent, multi-agent with a sliced map and multi-agent with intention sharing). Finally, we made several tests and analyzed results and performances as well as comparing the different approaches and examining possible future implementations. The code can be found on our GitHub repository [2].

Introduction

Deliveroo.js is a parcel-delivering web game. The goal is to pick up parcels and drop them at specific delivery zones to collect points in the form of parcels' values.

It can be played by humans but we were tasked to create an **autonomous software agent** that plays autonomously. Using a BDI (Belief-Desire-Intention) architecture, the agent will sense the environment, manage beliefs, deliberate intentions and execute plans to achieve its goals.

The game takes place in a grid-based environment, visible in Figure 1, with blocked tiles and walkable zones (that include the delivery cells, red in the picture). Parcels appear at different locations within the grid with a given value.

Players can move around the grid, pick up parcels, carry them, and put them down in any free tile. The objective is to maximize the number of **earned points** by delivering as many parcels as possible, with some scenarios including a decay time for their value (that we have to take into account).

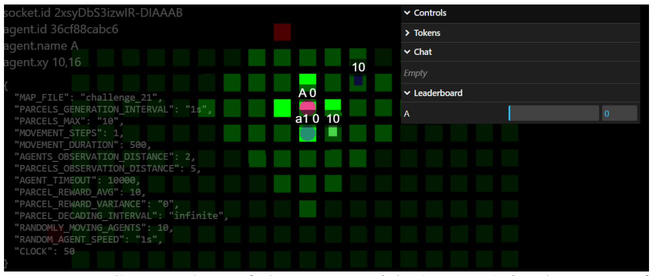
```
socket.id 2xsyDbS3izwIR-DIAAAB
agent.id 36cf88cabc6
agent.name A
agent.xy 10,16
{
  "MAP_FILE": "challenge_21",
  "PARCELS_GENERATION_INTERVAL": "1s",
  "PARCELS_MAX": "10",
  "MOVEMENT_STEPS": 1,
  "MOVEMENT_DURATION": 500,
  "AGENTS_OBSERVATION_DISTANCE": 2,
  "PARCELS_OBSERVATION_DISTANCE": 5,
  "AGENT_TIMEOUT": 10000,
  "PARCEL_REWARD_AVG": 10,
  "PARCEL_REWARD_VARIANCE": "0",
  "PARCEL_DECADING_INTERVAL": "infinite",
  "RANDOMLY_MOVING_AGENTS": 10,
  "RANDOM_AGENT_SPEED": "1s",
  "CLOCK": 50
}
```

Figure 1. Screenshot of the game with Agent A in the map of Challenge 21

In this project, we developed various approaches for autonomous software agents capable of playing the game. Also, two *multi-agent* approaches had been implemented and then analyzed in this report.

We integrated “automated planning” (with PDDL) into the agents' plan creation process, allowing them to generate plans to achieve their goals.

BDI architecture it's a cognitive model used in artificial intelligence that aims to represent an agent's reasoning and decision-making process by organizing information into beliefs (its own representation of the world), desires (goals), and intentions (the bridge between desires and actual actions).

1. Task Formalization

The architecture of the game is based on Node.js and Socket.IO, enabling communication between the client and server through websockets. The core game logic resides on the server.

After the connection to the server, we retrieved information (with the `onConfig` function) about the environment variables. We retrieved information like the **parcel observation distance** (for the *weightedBlindMove* function, explained later in the report), the **parcel decaying time**, the **movement duration** of the agent and the server's **clock**. We used them to compute the expected score for any parcel to actualize its value (the score that the parcel will have when the agent picks it up) in order to choose the best agent's intention.

As explained during the lectures we could consider “enemy” agents as permanent obstacles, elaborate a probability model for the trajectory of the enemies or with other strategies like game theory-based techniques. In our project, we decided to treat them as permanent obstacles so, when our agent creates the PDDL plan the “enemies” are considered as non-walkable cells. We first thought about a probabilistic model that takes into account where the enemy agent could be in the future, but due to the many possible implementations of an enemy agent, it could still lead to non-optimal decisions; this approach has been discarded in the brain-storming phase.

When our agents haven't any parcel (both carrying and in-memory) nor they don't sense any (its beliefs are empty) we force a *blind move*. This has been done to let the agent explore the game map when there are no profitable actions. Since the blind move is always a losing move (it doesn't produce any point) we wanted it to still be profitable. We weighted the points of the map so that the agent will choose to go to a further point from its position with a higher probability. This allows our agent to explore the map and increase the probability to find a parcel.

The weighted blind move works as explained in Algortihm 1.

Algorithm 1: Weighted blind move

```
Data: agentPosition
Result: A random weighted pair of point
offset = Math.ceil(config.get(“parObsDist”) / 2);
weight;
distances = [] ;
for $i = offset$ to $map.length - offset$ do
    for $j = offset$ to $map.length - offset$ do
        if $mapData[i][j] == 0$ then
            continue;
        end
        distance = Math.abs(agentPosition.x - i) +
          Math.abs(agentPosition.y - j);
        if $distance < offset$ then
            continue;
        end
        weight = distance / 5;
        for $k = 0$ to $weight - 1$ do
            distances.push({ x: i, y: j, score:
              Number.MIN_VALUE });
        end
    end
end
if $distances.length == 0$ then
    targetX = 0;
    targetY = 0;
    while $mapData[targetX][targetY] == 0$ do
        targetX = Math.floor(Math.random() *
          maxX);
        targetY = Math.floor(Math.random() *
          maxY);
    end
    return { x: targetX, y: targetY, score:
      Number.MIN_VALUE };
end
distances.sort(() => Math.random() - 0.5);
return distances[Math.floor(Math.random() *
  distances.length)];
```

Basically, we use the **parcel observation distance** of the agent to avoid selecting a point that is already in its FOV or that it’s too close to the borders of the map. Once we have the possible points (weighted in probability by the distance) we select a random one among them or, if the list of points is empty, we return a random walkable point in the map.

Testing Server We modified the `parcelGenerator.js` file in the given server repository removing the code that randomly spawns parcels. The custom server only spawns one parcel in a specific cell; we used it for testing purposes since in a controlled environment it was easier to track the decisions and actions of our agents.

2. Approaches

To approach this problem, we can describe three different macro-methodologies:

- **Hard-coded** solutions, where we explicitly tell the agent what to do in any specific case
- **Machine Learning** solutions, such as Reinforcement Learning where the agent learns a policy allowing it to choose for each state of the environment the action that maximizes the overall reward in a pseudo-trial and error approach
- **PDDL planning**, where we gather information about the world, we find a goal and we execute a plan in order to achieve such goal; the decisions are made by the agent itself based on an encoded strategy

In this project, we explored the last approach.

Our strategy is explained in detail in the following sections of the report, but in brief, we used the following weights/threshold in these specific cases:

- when deciding if picking up the “nth” parcel, we compute the score as “actualize return” described in the formula (1); in brief, it computes the current value plus the value of the nth parcel to pick up, and we discount it based on the time (and sequentially points) lost to pickup such parcel
- when choosing if it would be better to deliver (vs picking up) we decided to give a **virtual bonus** to that action with a $+25\%$ of the current value held by the player (to avoid situations in which the agent never delivers)
- when there is no parcels decay time, we force the agent to deliver when it reaches a certain number of parcels picked or if its distance from a delivery point is less than a threshold; we decided to map those values based on the *movementSpeed* of the agent so that a faster agent can wait more before delivering

$$\begin{aligned}
\text{score} &= currentScore + parcelReward \\
\text{lost} &= (\#parcelHeld + 1) \cdot moveDuration \cdot parcelDistance \\
\text{finalValue} &= score - \frac{lost}{parcelDecayTime}
\end{aligned} \tag{1}$$

To produce a plan, we encoded the map as a Matrix with 0 in cells that are blocked, 1 if they are walkable and 2 if they are delivery zones.

A* Before starting with the planning agent, we tried a basic agent where every case had to be hard-coded and we used the A* algorithm to compute the path to follow; we wanted to use this as the baseline for our project, but there were too many edge-cases and we decided to drop this approach since it was wasting a lot of time and giving poor results (file `single_agent_astar.js`)

PDDL Planning PDDL is a language used in artificial intelligence and automated planning to describe the specifications of a planning problem or domain. PDDL is designed to be a standardized representation that allows planners to understand and generate plans for a wide range of problems.

By default PDDL planning doesn't implement `types`, `or`, `not`, etc..., so we used the official documentation [3] to add the requirements needed for our project. In particular, we added the `typing, disjunctive-preconditions` and `negative-preconditions`.

3. PDDL Planning

For the PDDL planning, we started by defining the domain that contains actions and predicates. Our `domain.PDDL` file has six actions:

- **left** which allows our agent to move one cell left
- **right** which allows our agent to move one cell right
- **up** which allows our agent to move one cell up
- **down** which allows our agent to move one cell down
- **pickup** which allows our agent to pick up one or more parcels
- **putdown** which allows our agent to put down one or more parcels

We created prerequisites for movement actions: firstly our agent must be on the starting cell and the ending cell must be walkable, moreover, the destination cell must be free from any other agent on the map and must be a neighbour of the starting cell in the desired direction. For example, if our agent needs to move from the cell $c1 : \{x : 5, y : 5\}$ to the cell $c2 : \{x : 6, y : 5\}$, our agent must be in cell $c1$ and the cell $c2$ must be walkable, free from any agent and a `neighbourRight` of $c1$.

For the pickup action, we wanted our agent to be in the same cell as the parcel it wants to pickup, and also that the agent is not already holding such parcel; for the putdown action we want our agent to be holding the parcel it wants to deliver and to be in a delivery cell.

After that, we created the world definition and problem file. This procedure is divided into a static part and a dynamic one. The static part involves only the map representation which has been made with the Algorithm 2.

Where the `CheckNeighbours` function works as expressed in the Algorithm 3.

In this way, we were able to parse the map and create a PDDL problem that reflects the actual game scenario. We made this part static since the map shouldn't change during the execution; still, if that would be the case, we could just

Algorithm 2: Parsing the map as a PDDL problem

```
Data: map
Result: map represented in a PDDL string
for $i = 0$ to $map.length - 1$ do
    for $j = 0$ to $map.length - 1$ do
        switch $map[i][j]$ do
            case 0 do
                beliefsDeclare("is-blocked c_" + i + "_" + j);
            end
            case 1 do
                checkNeighbours(i, j, map);
            end
            case 2 do
                beliefsDeclare("is-delivery c_" + i + "_" + j);
                checkNeighbours(i, j, map);
            end
            otherwise do
                break;
            end
        end
    end
end
```

Algorithm 3: Check the neighbours of a cell

```
Data: x, y, map
Result: neighbours of a cell added to the beliefs
offsets = [
{ id: "Left", dx: -1, dy: 0 },
{ id: "Right", dx: 1, dy: 0 },
{ id: "Up", dx: 0, dy: 1 },
{ id: "Down", dx: 0, dy: -1 },
];
for offset of offsets do
    offsetX = x + offset.dx;
    offsetY = y + offset.dy;
    if $offsetX \geq 0$ and $offsetX < map.length$ and $offsetY \geq 0$ and $offsetY < map.length$ then
        if map[offsetX][offsetY] == 1 or map[offsetX][offsetY] == 2 and map[x][y] $\neq 0$ then
            beliefsDeclare("neighbour" + offset.id + " c_" + x + "_" + y + " c_" + offsetX + "_" + offsetY);
        end
    end
end
```

call the functions 2 and 3 every time we create the problem or when a plan fails (to keep our PDDL world consistent with the real world).

The dynamic part of the problem definition involves our agent, other agents and the parcels. We send all this information to the planner every time we want to create a plan since those values change very often. In this way, we always have an updated and reliable world to create our plans.

Lastly, the goal definition has been made with Algorithm 4.

Algorithm 4: Create a dynamic goal based on the desire of the agent

```
Data: desire, args, me
Result: PDDL goal for a plan
goal = "and";
if (desire == GO_PICK_UP) then
    goal += " (holding me_" + me + " p_" + args.id +
      ")";
end
else if (desire == GO_PUT_DOWN) then
    goal = "or";
    for a of args do
        goal += " (delivered p_" + a.id + ")";
    end
end
else
    goal += " (at me_" + me + " c_" + args.x + "_" +
      args.y + ")";
end
```

Moreover, we created some default objects like `a_default`, `p_default` and `c_default_default` to address cases in which some type of objects are not in the beliefs of the agent; these objects have no impact on the plans and are separated from the other cells and thus unreachable.

Once the problem is created we use the online planner `Planning Domains` [4] shown during the laboratories. Thanks to the API we were able to send our domain and problem to the planner which replies with a plan that satisfies our goal if there is one.

The *action flow* has been defined during the lessons and can be seen in the Figure 2. We implemented each block like this:

- **Environment**: it's the game world from which we obtain information to build our beliefs;
- **Belief revision**: during this step we use the functions `onYou`, `onParcelSensing` and `onAgentSensing` to generate and revise the agent's beliefs about the world;
- **Generate options**: once we revise our beliefs we generate all the options that the agent can follow;
- **Create intention**: with our scoring system, which will be explained later, we can select the best option among the ones generated and store the discarded ones in the agent's memory;
- **Select intention**: before starting with the new intention our agent performs an **intention revision** procedure in which it checks its memory to see if there are better intentions to follow, removing the ones that are too old;
- **Select plan**: once the intention is selected the agent checks if there is an old plan that can satisfy the new intention, if so, it applies such plan, otherwise it makes a call to the PDDL planner and a new plan is generated;
- **PDDL planner**: when it's called, it builds a dynamic problem, based on the current beliefs of the agent;
- **Plan execution**: when a plan is selected from the library or when it is returned by the planner, the agent starts to perform the actions; after a plan is successfully achieved, it is added to the plans library;
- **Intention replace**: During the execution of a plan the agent performs **intention replace** procedures; basically when a new intention is created during a plan the agent checks if such intention is better than the one it is trying to achieve; if so, the agent starts the plan related to the new intention, otherwise it continues with the original plan.

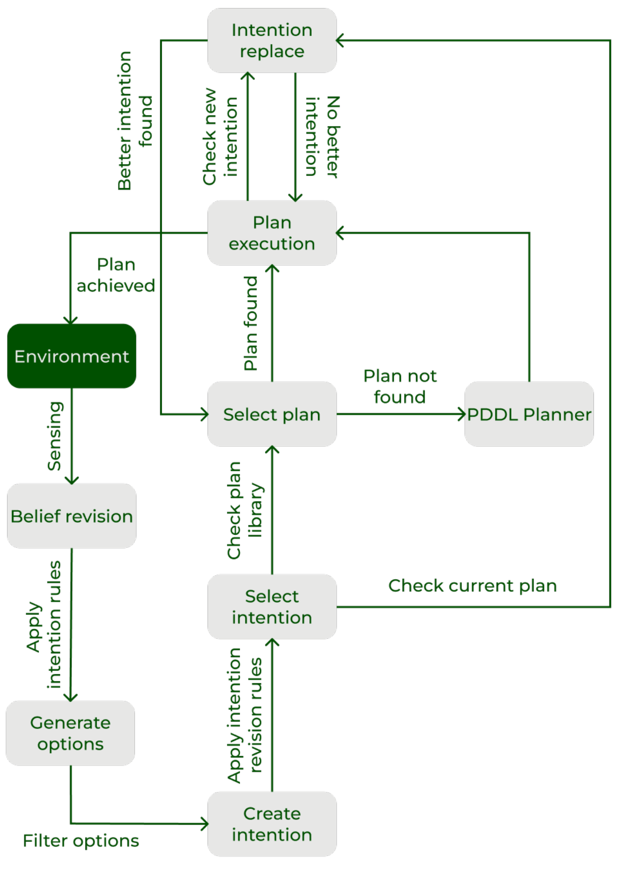

Figure 2. Action flow

3.1 Single-agent approach

We developed Agent A and we needed it to be able to collect parcels scattered in the environment and deliver them in the delivery zones. Agent A also needed to be aware of the world around it, it needed to know where are the parcels, where are the “enemy” agents and to be intelligent in the decision-making.

Agent A follows the action flow described in Figure 2. In particular, Agent A senses the environment with the provided functions; in this way, it knows where it is positioned on the map, where the parcels are and where the other agents are. After the sensing, the agent checks which actions can be done and it selects the best one. Agent A checks the possible options at every *clock of the server* (hard-coded value). In the meantime the agent keeps checking if new intentions are added to its queue and, if so, it checks if there is a plan that satisfies the current desire. If so, the agent creates the PDDL plan thanks to the planner (or it selects an already existing plan from the library) and starts executing it.

At every best option selected the agent checks if such option is better than the one it is achieving and, if that is the case, it stops the current plan and performs the new intention. Moreover, the agent has a **memory** of “discarded” options, this has been done since there may be cases in which an option is not optimal in the current situation but it could be the best later on. With the memory, the agent can keep track of the postponed options and, before starting an intention, it can check also the memory for any good option. The memory is constantly updated and options that are too old are dropped to avoid situations where our agent tries to achieve intentions that can’t be achieved at that moment (e.g. a parcel expired).

3.2 Multi-agent approach

The last required step in this project was to extend what we developed so far to a **multi-agent environment**.

Our objective was to create a team comprising two coordinated agents, namely Agent B and Agent C. To meet the basic requirements for this team, we developed a game strategy and coordination mechanisms, which will be elaborated upon in the report.

Both agents, B and C, will be equipped with the capability to **exchange information** about the environment, such as package locations and the positions of other agents. Additionally, they will exchange information about their mental states to ensure effective coordination of their activities.

Map slicing To divide the map into different slices, we implemented an algorithm that remembers the way a pizza can be cut to have a fixed number of slices with the same area. This concept has been extended to slice a 2D matrix.

It checks for each cell which slice it belongs to, based on the angle of the segment that connects the cell to the central point of the matrix. Then another for-statement is run in order to check if any cell in a slice is “isolated” and in such case, it gives that cell to the nearby slices. We can also select if we want to share the entire borders or only those “problematic” cells (by looking either at the up-down-left-right cells or also the one in the diagonals).

Any agent can deliver to any delivery point, to improve the overall performance.

The graphical representation of this algorithm can be seen in Figure 3 while the pseudocode can be seen in Algorithm 5.

We divide the map into two parts, for Agent B1 and Agent C1 as required, but our code can be easily extended to a number ≥ 2 of agents/slices.

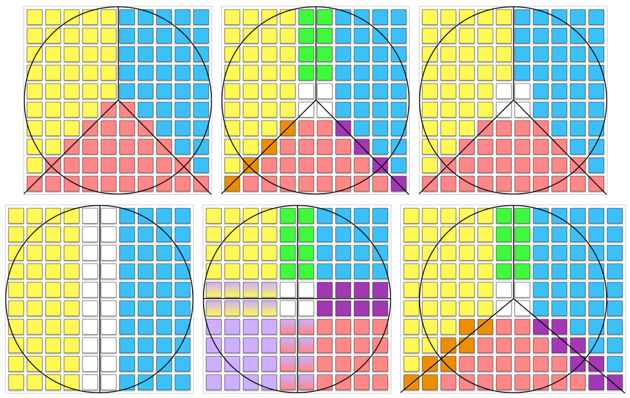
Figure 3. Extension of pizza slices to a square/rectangular 2D matrix

3.2.1 Exchange information about the environment

For the first part of the multi-agent approach, we implemented a basic **message-exchange procedure** that can be resumed like this:

1. each agent computes its own slice of interest
2. every action (described for the single agent) is restricted to its slice (including the `weightedBlindMove`)
3. during belief revision, if an agent senses a parcel and/or an agent (excluding the colleague) outside its own slice, it sends a message to the colleague using the `say` function:
    - messages for the parcels are encoded as follows: p\$p1.px.py.pcourier.preward_p2...
    - messages for the enemy agents are encoded as follows: a\$a1.ax.ay_a2...
4. the colleague can listen for messages using the `onMsg` function, checking if the sender ID is the one of the other “good” agent and calls the `messageParser` to reverse the encoding we made in the previous step
5. agents and parcels are then added to the list of saved agents and parcels (before adding the parcels, the agent checks if it is expired, computing in advance the time after which it will be expired)

3.2.2 Exchange information about mental states

In this approach, we also implemented the `ask` client function. The idea behind this implementation is to manage concurrent desires. If Agent B wants to pick up parcel $p1$, it asks Agent C

Algorithm 5: Dividing the matrix into slices

```
Data: numSlices, numRows, numCols, center, angle
Result: slices
for $i = 0$ to $numSlices - 1$ do
    slice = empty array;
    for $row = 0$ to $numRows - 1$ do
        for $col = 0$ to $numCols - 1$ do
            x = col - center[1];
            y = center[0] - row;
            currentAngle = atan2(y, x);
            if $currentAngle < 0$ then
                currentAngle += 2 * PI;
            end
            sliceStartAngle = angle * i;
            sliceEndAngle = angle * (i + 1);
            if $currentAngle \geq sliceStartAngle$ and
              $currentAngle < sliceEndAngle$ then
                slice.push([row, col]);
            end
        end
    end
    slices.push(slice);
end
for $cell\ of\ cellsInSlice$ do
    if $neighbouringCells\ in\ currentSlice \leq 3$ then
        add cell to neighbouringSlices ;
    end
end
```

the points it would gain from picking that parcel up. If Agent C’s current desire is to pick up the same parcel, it replies with the actualized values using the `averageScore` function; otherwise, it replies with MIN_VALUE. Agent B then checks whether its score is greater than C’s and eventually, it tells Agent C to stop its current intention; otherwise, it stops its own intention (to pick $p1$ up).

As before, we created a parsing function to analyze the received messages, allowing us to exchange information in a unique form (avoiding any “enemy” message).

The entire flow can be resumed in the following steps (apart from the first step, everything is done inside the *intentionCoordination* function):

1. **exchange ids**: both agents `shout` their own ids, until both acknowledge the other one
2. **pick up desire**: `ask` the other agent the expected score from its perspective for the same parcel
3. the second agent **replies** with `reply` with its expected score (or MIN_VALUE if its not its current desire)
4. the first agent **compares** the received value to its own value
5. if the first agent score is higher, it `says` the other to stop; otherwise it stops its own intention

To avoid parcels being ignored by the winning agent, we created two parcel sets (*whiteList* and *blackList*); each agent takes into consideration such sets when revising its beliefs.

4. Data Analysis

4.1 Data Acquisition

We run for 5 minutes Agent A in challenges 21, 22, 23 and 24 for 5 times each to reduce the random effect. Agent B1, C1, B2 and C2 have been run in challenges 31, 32 and 33 also for 5 minutes and five times.

During data acquisition, we saw how our agents work in the tough scenarios presented by the challenges. Overall, we saw that our agents were functioning correctly except in some edge cases. For example, challenge 33, presented a map divided into two big parts not linked together. So, our agents were forced to stay in the zone in which they spawned. This creates problems with the slicing of the map because one of the two agents has no way to reach its own slice and thus, to produce any plan. We forced the spawn of the Agents B and C to be one for each side.

4.2 Data Comparison

The results achieved by our Agents can be found in Table 1, Table 2 and Table 3.

Single Agent The single Agent A performed as expected and had no problems during the testing. It was able to achieve its goals consistently. As it’s possible to see in Table 1 Agent A met our expectations compared to the challenge 1 performed during the course.

SA	Run 1	Run 2	Run 3	Run 4	Run 5	avg
C21	290	310	260	300	360	304
C22	319	251	288	355	346	311.8
C23	2637	2398	2589	2613	2343	2516
C24	990	1138	855	909	1439	1066.2

Table 1. Single Agent Results

Multi Agent with map-slicing Agents B1 and C1 (the ones working with the “slices-coordination”) failed in the challenge 32 (Table 2). We understood that it was due to the fact that their basic-level communication was not enough to discriminate the best move (even if they were using the memory containing the paused intentions and the parcels to ignore). This resulted in them making a single move every couple of minutes (achieving zero points). In the other two challenges they behaved as expected, but due to the slicing function being very general, we think they could have got more points with a slightly challenge-based customized function.

Multi Agent Agents B2 and C2 had some problems in challenge 32 too: they sometimes had a slow start since the `weightedBlindMove` function generates points anywhere in the map, and when the plan fails they are removed from the possible outputs; as a result, the agents start with a list of possible points that are, for the most part, unreachable; with time, the agents clear the list and become more efficient; for this reason, their scores are very different across all the runs.

In challenge 31 Agents B2 and C2 behaved as expected, while in challenge 33 we noticed both very good and very bad runs; this was caused by the random location at the start and the fact that the maximum number of parcels spawned at the same time was 6. In some cases most of the parcels spawned only on one side of the map (or in cells that were not observed during blind moves), resulting in less points achieved.

5. Conclusions

Our project aimed to develop an autonomous software agent capable of playing the game from Deliveroo.js. Using a BDI architecture, our agents were designed to sense the environment, manage beliefs and intentions, and maximize the points earned by delivering parcels in the game.

Through the project, we explored various approaches, including single-agent and multi-agent strategies with both sliced map coordination and intention sharing. We implemented ”automated planning” using PDDL to enhance the agents’ decision-making process and generate plans to achieve their goals.

The single-agent approach, represented by Agent A, performed consistently and met our expectations. It successfully achieved its goals and maximized the earned points in all tested scenarios.

However, the multi-agent approach with map slicing, represented by Agents B1 and C1, encountered challenges in discriminating the best move due to limited basic-level communication. As a result, these agents achieved zero points in a specific challenge.

Agents B2 and C2, representing the multi-agent approach with intention sharing, faced initial challenges due to the `weightedBlindMove` function generating a large list of possible points. However, over time, they became more efficient and still performed well.

Comparing the different approaches, we observed that the multi-agent strategies exhibited both strengths and weaknesses. While some challenges were met as expected, others showed a wide range of performance due to random starting locations and the distribution of parcels.

Our results, presented in Table 1, Table 2, and Table 3, provide a comprehensive overview of the performance of each agent in different scenarios.

In summary, our project successfully developed autonomous agents that are able to play the Deliveroo.js game. We analyzed the strengths and weaknesses of the implemented approaches, highlighting areas for improvement such as enhancing communication and customization functions. We understood that our agents have limitations, but they represent a good starting point for the eventuality of future implementations. Overall, we think that we learned the foundation of autonomous agents and the logic behind it, and thanks to this project we also had a chance to test our knowledge to achieve its requirements.

5.1 Future Works

It could be interesting to explore alternative and various approaches for future implementations, such as:

- **Local planner**: implementing a local version of the PDDL planner to reduce the latency:
    - *pddl4j* [5] is Java native so it needed a bridge between the two languages, we tried to implement it but we discarded it quickly due to difficulties;
    - *jupypddl* [6] was no more supported so we avoided it;
    - *strips* [7] seemed perfect but it use some other syntax that was creating problem with our world definition so we discarded it since it was not compatible.
- **Reinforcement Learning**: developing a Reinforcement Learning model to later compare it with the other approaches
- **Enhanced communication**: exploring different communication methodologies

MAS	Run 1	Run 2	Run 3	Run 4	Run 5	avg
C31	497+955	785+746	706+824	820+976	927+903	1627.8
C32	0+0	0+0	0+0	0+0	0+0	0
C33	708+594	545+750	813+997	949+807	527+898	1517.6

Table 2. Multiple Agent Results Slicing

MAM	Run 1	Run 2	Run 3	Run 4	Run 5	avg
C31	830+657	679+677	715+713	643+969	1122+883	1577.6
C32	1131+1247	854+999	361+226	961+442	997+975	1638.6
C33	589+417	691+421	215+712	424+700	332+498	999.8

Table 3. Multiple Agent Results

References

[1] Deliveroo server code. [Online]. Available: https://github.com/unitn-ASA/Deliveroo.js

[2] GitHub repository with the project. [Online]. Available: https://github.com/davidemodolo/ASA_Delivery_Bot

[3] PDDL Documentation. [Online]. Available: https://planning.wiki/

[4] Online planner used to compute PDDL plans. [Online]. Available: http://planning.domains

[5] PDDL4J Repository (Java). [Online]. Available: https://github.com/pellierd/pddl4j

[6] JUPYDDL Repository (Python). [Online]. Available: https://github.com/apla-toolbox/pythonpddl

[7] STRIPS Repository (Javascript). [Online]. Available: https://github.com/primaryobjects/strips