GlovoJS: a BDI based solution to play the dynamic parcel delivery game ”Deliveroo.js” with cooperative autonomous agents

Edoardo Meneghini[1], Lorenzo Orsingher[2]

Abstract
This project, under the name of GlovoJS, explores the development of an autonomous multi-agent system designed to play Deliveroo.js, a game simulating the logistics of courier services. In artificial intelligence, an agent perceives the environment it is in through sensors and acts upon it using actuators, striving to optimize its performance based on available knowledge and resources. Beyond rationality, agents must also exhibit autonomy, learning and adapting independently from external control to address incomplete or incorrect prior knowledge. In Deliveroo.js, agents operate on a two-dimensional map where parcels are randomly generated, each with a reward value that decays over time. The objective for the agents is to maximize their total reward by efficiently collecting and delivering these parcels to specified locations. The game introduces additional challenges through various constraints and limitations that the agents must navigate. This project aims to create a sophisticated multi-agent system based on the BDI approach and with the integration of PDDL. Such system is capable of effectively tackling these challenges, thus providing insights into the application of AI in a simplified, yet realistic version of a real-world problem.

Keywords
Autonomous Software Agents — Wayfinding — Collaborative Agents

[1] *M.Sc. in Computer Science, University of Trento, edoardo.meneghini@studenti.unitn.it*
[2] *M.Sc. in Artificial Intelligence Systems, University of Trento, lorenzo.orsingher@studenti.unitn.it*

Contents

1. Introduction

In the last few decades, due to the introduction of embedded electronics in many market segments and in most of the things we interact with in our everyday life, the need for systems that are able to identify their goals, know the set of rules they're supposed to follow and plan in order to achieve goals, is more important than ever. While many systems are deeply interconnected between each other and don't need to be autonomous, some others can't afford to. One such example is autonomous vehicles fleets. Each car can exchange data with other vehicles, or receive instructions from a machine learning algorithm running in the cloud. However, vehicles need to be able to operate autonomously, even when not connected to the internet, or to other agents. Building such agents can be very different from what we're used to. As a matter of fact, one of the most widely used architectures in the development of autonomous software agents is BDI, which stands for "Beliefs, Desires, Intentions". The main idea behind BDI is that the reasoning scheme of the agent should be structured around these three concepts. Beliefs are the internal representation of the world that agents have managed to sense. Desires represent the goal of the agents in terms of what they would like to achieve eventually and Intentions are the actuators that can get the agent from desiring a certain goal, to achieving it. In the field of autonomous agents, this whole architecture can be implemented seamlessly with the use of PDDL planning, which is the heart of how the system encodes this BDI structure, defines a domain, bundles and sends a request to a solver, and parses the response, translating it into actual actions for the agent.

In this report, we're going to discuss how we have developed a multi-agent system called GlovoJS, designed to efficiently play the Deliveroo.js game in full autonomy.

2. Game, setup and tools

2.1 Outline of the game

Deliveroo.js is a 2d grid-based parcel delivery game, designed to provide an interactive environment where agents can compete to achieve the highest score, while developing and solving plans dynamically. At the beginning of the game, agents are placed in a map that is represented by a grid, where every tile is either walkable or non-walkable. Walkable tiles can be delivery zones, or parcel spawning tiles. On each parcel spawning tile, at random intervals, parcels can appear. Every parcel has a certain score assigned to it, which may or may not decay at a given rate, according to the configuration of that specific game. Agents that want to pick up a parcel need to move to that parcel's tile first, and then perform a pick up action. Same goes for the delivery of parcels. Agents can carry as many parcels as they want, but something to be kept in mind is that they are only awarded the points corresponding to the value of a parcel at the time of delivery, and parcels expire as soon as their value reaches zero. The game can be played by single agents competing in a free-for-all fashion, or by multiple fleets of agents that play as teams. The goal of the game is to get the highest score among all players in the map.

At the beginning of every round, a config file is sent to all clients. This file contains a matrix representing the map and all arbitrary parameters (movement speed, average parcel value, parcel decay rate, etc.) are set.

2.2 Tools and technologies

Tools used in the development of the project include, but are not limited to:

- JavaScript and the Node environment for the overall implementation of the agents system and communication with the APIs.
- Deliveroo API[1] is the package that enables interactions between the client and the server, while also providing the key primitives (move, pickup, putdown) that agents will make use of.
- Planning Domains Editor[2] is an online environment with capabilities to write, edit, validate, parse and solve code written in PDDL. While not providing great performance when used remotely as a solver, it turned out to be a fundamental tool of our pipeline.
- The `Planutils`[3] repository turned out to be particularly useful because, as discussed further in the report, using only the online solver would have been a significant bottleneck. Setting up a local docker-based PDDL solver helped us a ton reducing delay and increasing overall reliability and performance.
- `@unitn-asa/pddl-client`[4]: this package provided all the abstractions needed to streamline the interaction with both the online and the local PDDL solver, managing and parsing PDDL strings, requests and responses.

2.3 File structure and setup

The setup instructions and detailed explanations about the working components of the project can be found both in the ReadMe file and throughout the code. Attached below there is a scheme of the file structure, that will be explained in further detail in the next section.

File/Directory	Description
GlovoJS/	
configs/	
custom/	
`solver_custom.js`	Custom PDDL solver integration
`solver_bkp.js`	Backup online PDDL solver
dashboard/	
`dashboard.html`	Dashboard UI
`dashboard.css`	Dashboard styling
`server.js`	Dashboard server setup
data/	
`action.js`	Defines Action class and types
`field.js`	Game field and pathfinding
`position.js`	Position class and directions
`tile.js`	Individual tiles in the field
planner/	
`bfs_pddl.js`	PDDL integration for pathfinding
`plan_cacher.js`	Caching system for plans
`agent.js`	Main agent logic and game loop
`geneticBrain.js`	Genetic algorithm for planning
`rider.js`	Rider class used by game agents
`utils.js`	Utility functions
`index.js`	Entry point
`README.md`	Project documentation

3. System Architecture

The system is structured so that each component is modular, can be easily replaced or extended and seamlessly interacts with the other components. Such architectural decision was made to ensure that the system is scalable, maintainable and easy to debug. Although there are some drawbacks to this approach, such as the increased complexity for some tasks, the benefits outweigh the costs.

GlovoJS is composed of several main components, each with its own responsibilities. The main components are Agent, Rider, Brain, Field and Dashboard. The Agent class is responsible for interacting with the game server, receiving the game state and sending the actions to the server. In this class are present all the sensing functions that allow the agent to perceive the game state and the main loop for the riders. The Rider class represents the agent that moves on the grid, picks up and delivers parcels. Each rider contains all the information about its game state, including its entire percept and the plan it is following. The Brain class is responsible for generating the plan that the rider will follow. The plan is generated in two main steps, first the set of possible routes are computed using the pathfinding planner of choice, then using this information the genetic algorithm creates the best plan for each rider, maximizing the expected score. The Field class represents the game field, it contains all the information about the parcels, delivery zones and spawn zones. This information is used to generate the graph that is used by planner. The Dashboard class offers a real-time visualization of the game state and the agents' actions. This tool is extremely useful for debugging the system and understanding the behavior of the agents.

3.1 BDI architecture

The Belief-Desire-Intention (BDI) structure is a widely-used framework for designing intelligent agents [6], and while not explicitly named as such, this exact structure has been at the heart of the implementation of autonomous pickup and delivery agents in this project.

Beliefs: The belief component of the BDI structure is primarily represented by the agentsBeliefs, mapBeliefs and parcelsBeliefs, from within the Field and the Agent class. The Field class maintains a representation of the game environment, including the map layout, walkable tiles, delivery zones, and parcel spawners. All of these make up the agent's beliefs about the state of the world. The Beliefset is used to store and manage predicates about the environment, such as the connectivity between tiles and the presence of parcels or agents at specific locations. These beliefs are continuously updated as the agent receives new information through its sensing capabilities provided by the Deliveroo.js API, implemented via various event listeners in the Rider class (e.g., onParcelsSensing, onAgentsSensing).

Desires: The desires in this system are implicitly represented by the overall goal of maximizing the score by efficiently collecting and delivering parcels. This high-level desire is encoded in the fitness function of the Genetic class, which evaluates potential plans based on the rewards they would yield. The genetic algorithm's population represents a set of candidate desires (potential plans) that the agent considers.

Intentions: The intentions are manifested in the plans generated by the Genetic class and executed by the Rider class. The genetic algorithm in the Genetic class generates a sequence of actions (represented by the Action class) that the agent intends to carry out. These intentions are not fixed but can be revised based on new information or changing circumstances. The planLock mechanism in the Genetic class ensures that intention reconsideration (replanning) doesn't occur too frequently, providing a balance between commitment to current intentions and flexibility to adapt to changes. The BDI cycle is implemented in the main loop function in the masterAgent.js file. This loop continuously checks the current state (beliefs), generates plans based on the current goals (desires), and executes the chosen plan (intentions). The cycle includes mechanisms for handling unexpected situations, such as blocked paths or disappearing parcels, which trigger replanning.

3.2 PDDL Integration

PDDL (Planning Domain Definition Language) plays a fundamental role in our project, as it is the core component of the pathfinding and planning mechanism. Its goal is primarily to optimize the movement and actions of multiple agents in the game environment. The integration is centered around the `path_planner.js` file, which contains functions for dynamically generating PDDL domain and problem descriptions based on the current game state, which is comprehensively represented in the BeliefSet. These descriptions include the positions of agents, parcels, and obstacles, as well as the connections between different locations on the game map. The `parellel_pddl` function allows for solving multiple pathfinding problems simultaneously, potentially improving efficiency when planning for multiple agents. The PDDL solver is used as an alternative to traditional pathfinding algorithms, with the Field class in `field.js` having a `USE_PDDL` flag to toggle between PDDL-based and traditional pathfinding methods. This integration allows the project to leverage the power of autonomous planning techniques, potentially leading to more optimal and complex plans for the agents, especially in scenarios involving multiple agents and dynamic obstacles, which are exactly what the Deliveroo.js game is about.

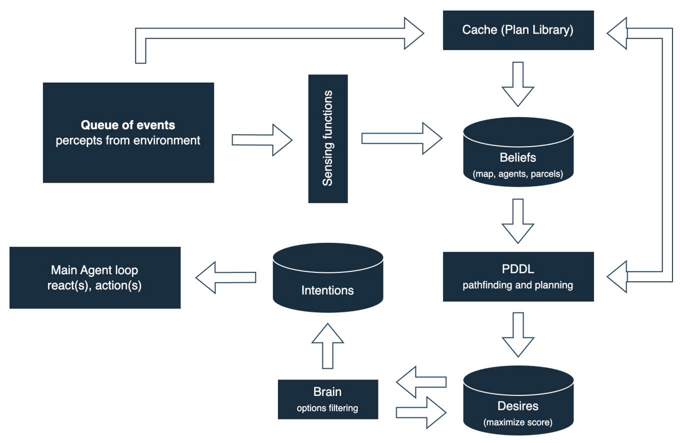


Figure 1. *Scheme representing a high level overview of the BDI scheme adopted, showing the components it involves*

4. Algorithms and Implementation

4.1 Path Planning

In the context of multi-agent parcel delivery systems, choosing the right pathfinding algorithm plays a crucial role in overall system performance. Breadth-First Search (BFS) emerges as a particularly suitable choice for this scenario, offering great balance of simplicity, efficiency, and effectiveness. BFS's strengths lie in its fundamental characteristics. In an unweighted graph or grid, which is often the case in simple game-like environments, BFS guarantees finding the shortest path. This optimality is crucial in a delivery scenario where efficient routes directly impact performance. Moreover, BFS is remarkably straightforward to implement and understand, making it easier to maintain and debug – a significant advantage in complex multi-agent systems where other aspects of the algorithm may demand more attention. The memory efficiency of BFS is another point in its favor. In a grid-based environment, BFS can be implemented without needing to store the entire graph in memory, a consideration that becomes increasingly important as the scale of the environment grows. This efficiency extends to its predictable time complexity $O(V+E)$ where V is the number of vertices and E is the number of edges, or $O(n)$ in a grid where n is the number of cells. This predictability allows for better overall system planning and resource allocation. Importantly, BFS offers completeness, always finding a solution if one exists. In a delivery scenario where ensuring reachability is crucial, this characteristic provides a valuable guarantee. Furthermore, BFS can be easily adapted to handle dynamic obstacles, such as other agents, by rechecking node availability during the search. This adaptability is particularly valuable in a multi-agent system where the environment is constantly changing.

4.2 Decision Making

The pickup-delivery problem in Deliveroo.js is complex and has no optimal trivial solution, the task is similar to the one of VRPPD (Vehicle Routing Problem with Pickup and Delivery) [5] which is a variation of the TSP (Traveling Salesman Problem), a well-known NP-hard problem. The presence of multiple agents and score-decaying parcels further complicates the task, requiring a sophisticated decision-making process to optimize delivery plans.

After testing out different approaches, we opted for the idea of using a **genetic algorithm** to develop an optimization system as the brain for our planner. This approach combines elements of graph theory, genetic algorithms, and heuristic search to coordinate the work of multiple components of our code, with the ultimate goal of finding efficient delivery plans for multiple agents simultaneously. It's been shown that Genetic Algorithms can tackle this kind of complex tasks very efficiently [7] by mimicking some of the processes of natural selection that belong to the larger class of evolutionary algorithms (EA).

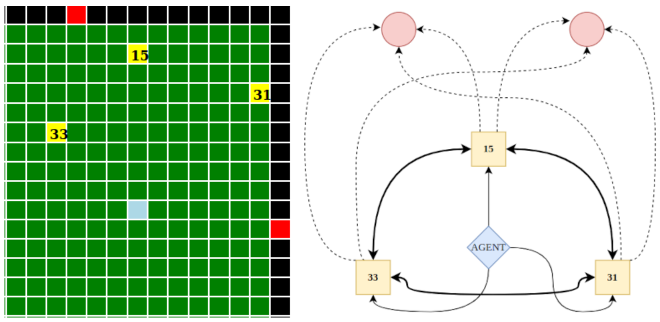


Figure 2. *The game state on the left is abstracted to the graph representation on the right. Dotted lines represent the routes from each parcel (in yellow) to all the delivery zones (in red), thick lines are the routes between parcels and the thin lines are paths from the agent to the parcels*

The algorithm begins by **constructing a graph representation of the environment** (Figure 3), this step is not only important, it's also the most computationally intensive one. The graph is constructed by considering the current belief state of the agents, the path planner is asked to provide a set of possible routes between the parcels, agents and delivery zones, these routes are then used to build a graph: nodes represent parcels, delivery zones and agents, while edges represent paths between these locations. The weight of each edge is determined based on the distance between locations and the current state of the environment, including factors such as blocking agents. This graph serves as the foundation upon which the optimizator operates. At the center of our program there is a genetic algorithm that evolves potential solutions, or delivery plans, over multiple generations, the idea is to provide multiple possible route options and let the genetic algorithm find the best combination in order to maximize the expected delivery score. Each solution is encoded as a *family* of *DNA* sequences, where each *DNA* sequence is a series of *genes* representing the order of parcel pickups for a single agent. The algorithm maintains a population of these families and evolves them over time. The evolution process involves several key operations.

1. **A multi-crossover operation** allows for the exchange of partial plans between agents, enabling the algorithm to explore combinations of successful strategies across different agents.
2. **Random mutations** are introduced to maintain diversity in the population, preventing premature convergence on suboptimal solutions.
3. **The selection of parents** for the next generation is performed using a roulette wheel method, where fitter solutions have a higher probability of being selected, thus driving the population towards better solutions over time.

Central to the genetic algorithm's success is its **fitness function**. This function evaluates the quality of each solution by considering a range of factors. It takes into account the total reward from delivered parcels, the cost of movements, the decay of parcels over time, and the current carrying capacity of agents. By incorporating these various elements, the fitness function provides a nuanced assessment of each solution's effectiveness, allowing the algorithm to optimize for complex, multi-faceted objectives. One of the key strengths of the geneticBrain approach is its adaptability. **The algorithm can dynamically adjust plans based on changes in the environment**, such as new parcels appearing or other agents blocking paths. This adaptive planning capability makes it particularly well-suited for the Deliveroo.js game, where conditions are constantly changing either due to other players, or different map configurations. Another key advantage of the fitness function is that it uses a simple *stepCost function* to calculate the cost of each step. Therefore, just changing the weights of this function is enough to alter the behavior of the agents, making them more or less aggressive in their delivery strategy. It's also trivial to add new factors to the stepCost function in order to accommodate new requirements or constraints. Moreover, since the genetic algorithm is agnostic to the number of agents sharing information, **this approach naturally optimizes plans for multiple agents simultaneously**. This leads to more globally optimal solutions, as it can consider the interactions and potential conflicts between different agents' plans. It is also inherently prone to balancing between exploration and exploitation, since it allows the system to discover innovative delivery strategies that might be overlooked by greedy or purely heuristic approaches. This balance enables the algorithm to escape local optima and potentially find superior solutions in complex solution spaces. Lastly, we strongly feel that **scalability is another strength of this approach**. The algorithm can handle a varying number of agents and parcels without significant changes to its core structure, making it adaptable to different problem sizes. This scalability, combined with the algorithm's ability to consider complex factors in its fitness function, allows for more nuanced decision-making compared to simpler heuristic approaches. Perhaps most importantly, the geneticBrain algorithm excels at **long-term planning**. Unlike greedy algorithms that might focus on immediate gains, this approach can optimize for long-term rewards. By considering the future implications of current actions, it can potentially lead to better overall performance over extended periods of operation. The entire procedure is illustrated in Algorithm 1.

Algorithm 1 Genetic Algorithm for Score Maximization

```
1:  Initialize $population \leftarrow [pop\_size]$
2:  for $i \leftarrow 0$ to $gen\_num$ do
3:      $new\_pop \leftarrow []$
4:      $chances \leftarrow \text{rouletteWheel}(population, graphs)$
5:      $elites \leftarrow \text{getElites}(population, chances, elite\_rate)$
6:      $new\_pop \leftarrow new\_pop.concat(elites)$
7:      for $j \leftarrow 0$ to $pop\_size - elites.length - 1$ by 2 do
8:          $parentA \leftarrow \text{pickOne}(population, chances)$
9:          $parentB \leftarrow \text{pickOne}(population, chances)$
10:         $childA \leftarrow \text{multiCrossover}(parentA, parentB)$
11:         $childB \leftarrow \text{multiCrossover}(parentB, parentA)$
12:         $new\_pop.\text{push}(childA)$
13:         $new\_pop.\text{push}(childB)$
14:     end for
15:     for $family$ of $new\_pop$ do
16:         if random() $< mutation\_rate$ then
17:             $family \leftarrow \text{mutate}(family)$
18:         end if
19:     end for
20:     $population \leftarrow new\_pop$
21:     $tot\_fit \leftarrow 0$
22:     for $family$ of $population$ do
23:         $fit \leftarrow \text{fitness}(family, graphs)$
24:         $tot\_fit \leftarrow tot\_fit + fit$
25:         if $fit > best\_fit$ then
26:             $best\_fit \leftarrow fit$
27:             $best\_dna \leftarrow family$
28:         end if
29:     end for
30: end for
```

The genetic algorithm provides much flexibility and provides numerous parameters that can be tuned to optimize performance:

- **population size:** the number of families in the population, larger populations can lead to better solutions but require more computational resources, for maps with a large number of parcels and agents a larger population is recommended. Empirically we found that a population size of 100 is a good compromise between performance and quality.
- **generation number:** the number of generations the algorithm will run for. More generations may allow for better solutions but also require more computational resources and time. With our set of parameters we found that in 30 generations the algorithm converges to a good solution and fitness doesn't improve significantly after that.
- **mutation rate:** is the probability that each *family* will be subject to mutation. It encourages diversity and can prevent getting stuck in a local minima.
- **elite rate:** the percentage of the population that is considered elite and is passed unchanged to the next generation. A *family* is considered elite if its fitness is in the top *elite_rate* percent of the population.

4.3 Multi-Agent Coordination

For multi-agent coordination there are two main possibilities: centralized and decentralized approaches. In the centralized approach, a single agent is responsible for coordinating the actions of all agents, while in the decentralized approach, each agent makes its own decisions based on local information. For our project we opted for a centralized approach and we decided to use an external entity (called Brain) to coordinate the actions of the agents.

Every time an agent senses a change in its belief state (e.g. a new parcel appears, an agent blocks his path, etc.), it communicates to the Brain the need for a replan. At this point the Brain collects the current state of the environment from each agent, generates a new plan and, if the fitness of the newly generated plan is better than the previous one, it sends the new plan to the agents. This approach allows for a more efficient coordination of the agents, as the Brain can take into account the global state of the environment and generate plans that are optimized for the entire fleet of agents. Moreover, it allows for a more efficient use of the PDDL solver, as the Brain can bundle multiple requests in a single call. Another key advantage of our approach is that the planner can work with any arbitrary number of agents, distributing the parcels among them in an optimal way regardless of the characteristics of the map. Additionally, the complexity of planning only increases linearly with the number of agents.

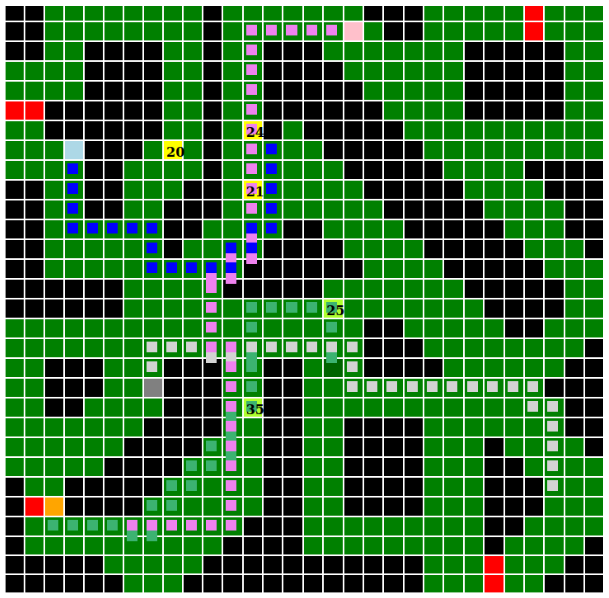


Figure 3. *A dashboard view of 4 agents cooperating. Each coloured path corresponds to an agent plan, light green tiles are parcel to be picked up and orange tiles are designed for the plan delivery.*

A key aspect of multi-agent cooperation is that no two plans can contain the same parcel. This hard constraint is implemented inside the genetic algorithm where if the same gene (parcel) is found in multiple DNAs (plans) within the same family (collection of plans for all agents), the family gets discarded, both in the crossover operation and in the mutation step,

While a large number of agents allow for a more efficient delivery of parcels and a more comprehensive coverage of the map, it comes with its own set of challenges. As the number of agents increases the amount of calls for replanning follows the same trend, replanning is a computationally intensive operation and each change of plan is expensive and introduce indecisiveness in the agents, for this reason we introduced a lock mechanism that prevents agents from replanning when a new generation is being computed, additionally we introduced a timeout mechanism that avoid too many calls to the Brain in a short period of time and, as a final measure, new plans are only accepted if they are better than the previous one by a certain threshold.

5. Performance Optimization

5.1 Parallelization

The PDDL solver is at the core of our planning strategies, as it takes care of generating the routes followed by the agents and it is tasked with the most computationally intensive part of the system. However, we have quickly realized that the solver alone is not fast enough to handle the amount of requests that we need to make in real time and acts as a bottleneck. Even while using the local dockerized version of the solver, the overhead due to the communication between our system and the API is too high.

To mitigate this issue, we have introduced a parallelization mechanism that allows us to bundle multiple requests in a single problem and send them to the solver in a single call. There are three components in our code that which make heavy use of the pathfinding algorithm, this means these sectors are also potential candidates for the implementation of parallelization.

- Computing routes from the agent to all parcels in sight
- Computing routes from each parcel to all delivery zones
- Computing routes from each parcel to all other parcels

A single pathfinding task is defined as a *couple* (as in a couple of *initial state* and *goal position*), before sending the request to the solver, we bundle multiple couples in a single list and while building the PDDL problem we set multiple starting states and multiple goal states, the solver will then return a list of paths, one for each couple. This approach allows us to reduce the number of calls to the solver by a factor equal to the number of couples in the list. Unfortunately, this approach does not solve all the problems and introduces two issues:

- If even only one of the goals is not reachable, the whole problem is considered unsolvable and the solver returns `false`. This means that whenever a parallelized task fails, we need to fall back to the standard approach and sequentially compute the paths. This introduces a significant overhead and slows the entire process down. However, the benefits of parallelization still outweigh the costs in most cases.
- The complexity of the problem grows exponentially with the number of goals. Although the overhead due to the communication with the solver makes single calls unfeasible, as the number of goals reaches a certain threshold (around 8, but it depends on the map) parallelizing tasks becomes more expensive than doing them sequentially, as shown in Figure 5. For this reason – and also to avoid the issue mentioned above – we have introduced an additional mechanism inside our PDDL planner that takes care of splitting large problems coming from the Brain into smaller *chunks* that are then sent to the solver.

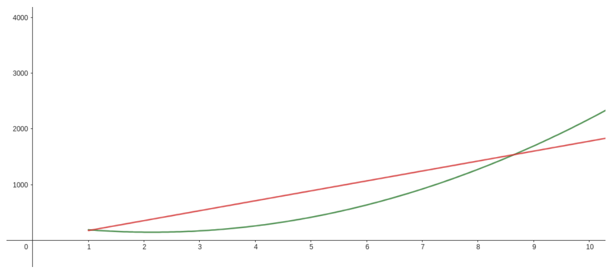


Figure 4. *On the x axis the number of goals and on the y axis the ms to compute the problem. The green curve is fit to follow the trend of parallelized solving meanwhile the red curve shows the amount of time needed to compute the same number of goals sequentially. It's possible to see how after around 8 goals it's not convenient anymore to parallelize.*

A closer look at the solver showed how, in the case of a single goal, it takes only few milliseconds to compute the problem. However, over 140*ms* are added just by sending the request and waiting for the response. This leads us to believe that most of the performance issues could be resolved by using an **integrated solver** locally.

5.2 Caching strategies

The presence of well-defined delivery zones and parcel-spawning tiles in the game environment allows for the implementation of a caching system that can significantly reduce the computational load of the pathfinding algorithm. The *plansCache* class is responsible for the storage and retrieval of precomputed paths between locations in the game map. Unfortunately, it's not enough to store the paths between tiles, since external agents can block paths and effectively make tiles non-walkable.

Cache entries are stored in a quick-access *Map* where the entry key is a string concatenation of the two locations' coordinates followed by the coordinates of all the blocked tiles reported during the pathfinding computation, this way each entry is a unique identifier of a specific game state. To further optimize the cache, when checking for a path, we look both for the requested direction and the inverse direction, this way we can avoid recomputing the same path twice. If a hit for the inverse entry is found, we can simply reverse the path and return it.

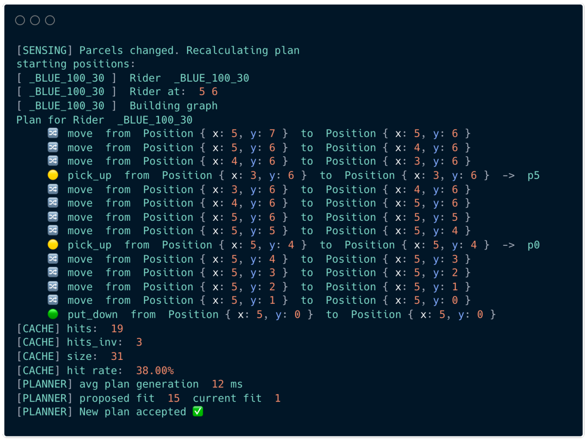
Assessing the effectiveness of the caching system is a difficult task, as it's heavily dependent on the specific game state and the map configuration, however empirical evidence showed cache hit rates that range from 50% in the most chaotic, large and unpredictable maps to over 99% in the most structured and predictable ones. Needless to say it significantly improves the performance when paired with the PDDL path planner as every call to the solver matters. We have also introduced a so called ”*boost*” mode that uses the first few seconds of the game to precompute all the paths between the tiles. By doing so, the cache is already populated when the agents start moving.

6. Evaluation and Results

6.1 Performance metrics

Developing an autonomous software agent inside the scope of this project can be challenging for a number of reasons. First of all, there are many components that are deeply intertwined and need to work perfectly together like a clockwork, where each of the gears needs to be synchronised with the others. While this may sound easy, things can get messy quickly when working with many asynchronous functions that send, request and receive data hundreds of times per second. Moreover, different maps test the agent's resilience to different problems, and many other challenges arise with the introduction of "enemy" agents, non-discrete positions, communication systems, etc. Debugging errors and potential issues can be very difficult, therefore we have implemented three main systems to do so:

- **Dashboard:** we have developed a GUI that served as a real-time dashboard. It renders a 2d representation of what is happening on the map, showing agents, their current plan (if said agents are friendly), what parcels they are carrying, what parcels they want to deliver/pickup, what enemy agents they're trying to avoid, etc.
- **Analytics:** we have added time and memory usage logging in multiple parts of the code to keep track of how many resources we're using and where. These analytics have proved fundamental when looking for bugs and inefficiencies, as well as timeout issues. We have also used this empirical observations as a basis for decision making in specific implementation choices, such as balancing parallelization with problem size, that we have discussed in this report.
- **Enhanced Logs:** Printing logs to console became very quickly one of our most useful debugging tools, since it allowed to keep track of the sequence of events that led to good or bad scenarios. We have kept refining the style and format of logging functions to achieve maximum clarity and density of information.



```
[SENSING] Parcels changed. Recalculating plan
starting positions:
[ _BLUE_100_30 ]  Rider  _BLUE_100_30
[ _BLUE_100_30 ]  Rider at:  5 6
[ _BLUE_100_30 ]  Building graph
Plan for Rider  _BLUE_100_30
     🔀 move  from  Position { x: 5, y: 7 }  to  Position { x: 5, y: 6 }
     🔀 move  from  Position { x: 5, y: 6 }  to  Position { x: 4, y: 6 }
     🔀 move  from  Position { x: 4, y: 6 }  to  Position { x: 3, y: 6 }
     🟡 pick_up  from  Position { x: 3, y: 6 }  to  Position { x: 3, y: 6 }  ->  p5
     🔀 move  from  Position { x: 3, y: 6 }  to  Position { x: 4, y: 6 }
     🔀 move  from  Position { x: 4, y: 6 }  to  Position { x: 5, y: 6 }
     🔀 move  from  Position { x: 5, y: 6 }  to  Position { x: 5, y: 5 }
     🔀 move  from  Position { x: 5, y: 5 }  to  Position { x: 5, y: 4 }
     🟡 pick_up  from  Position { x: 5, y: 4 }  to  Position { x: 5, y: 4 }  ->  p0
     🔀 move  from  Position { x: 5, y: 4 }  to  Position { x: 5, y: 3 }
     🔀 move  from  Position { x: 5, y: 3 }  to  Position { x: 5, y: 2 }
     🔀 move  from  Position { x: 5, y: 2 }  to  Position { x: 5, y: 1 }
     🔀 move  from  Position { x: 5, y: 1 }  to  Position { x: 5, y: 0 }
     🟢 put_down  from  Position { x: 5, y: 0 }  to  Position { x: 5, y: 0 }
[CACHE] hits:  19
[CACHE] hits_inv:  3
[CACHE] size:  31
[CACHE] hit rate:  38.00%
[PLANNER] avg plan generation  12 ms
[PLANNER] proposed fit  15  current fit  1
[PLANNER] New plan accepted ✅
```

Figure 5. *Example of a terminal output that shows some of the logging before and after a plan is found, including the plan itself and it's styling that allows for immediate visual pattern matching.*

6.2 First challenge results

The first challenge – that implied the use of a single agent – showed us promising results for the overall project and gave us confidence to continue working on our idea. We scored 2nd overall and our agent, while not being exceptional in any case, showed robustness and flexibility in all maps. At the same time, we have noticed a number of things that could be improved, mainly centered around the interaction with other agents. We have quickly translated these weaknesses into actionable tasks that were promptly fixed before moving on towards the implementation of PDDL.

6.3 Comparison between standard bfs and PDDL

Using standard BFS allows for great execution speed, since the algorithm itself is designed to solve exactly the problem that we're looking into with our project: finding if two nodes in an undirected unweighted graph are connected and, if so, find the shorter path between from one to the other. We have noticed that this approach requires just some small precautions in order to work seamlessly. Moreover, we can manage to compute hundreds of BFSs calls in little-to-no time. On the other hand, the PDDL-based path planner takes a considerably larger amount of time. The algorithm per se is not responsible for this increased time – the overhead is, as discussed in the Performance Optimization section.

7. Benchmarking

In order to evaluate the performance of our system with various configurations and parameters, we have conducted a series of benchmarking tests. These tests aimed to assess the system's efficiency, scalability, and robustness under different conditions. The primary metrics used for evaluation were the **points per minute (PPM)**, taking the measure at different points during the game ensured to have a more comprehensive view of the system's performance, including the long-term effects of components such as the caching system.

Agents have many parameters and are heavily customizable: the main differences between the agents will be the Path Planner of choice (PDDL or BFS) and the use of the search boosting mechanism. While different combinations of the genetic algorithm setting do impact the performance of the agents, they are less significant then the above-mentioned ones. The 3 agents of choice are:

- **ALLBFS:** This is thought to be the best performing configuration, it uses the fast BFS path planner without the search boosting mechanism. Just like the other agents, population size is set to 100 and the number of generations is 30. This agent has no limit for what concerns the number of parcels and delivery zones considered in the graph construction.
- **ALLPDDL:** In order to get competitive speed from the agents using the PDDL solver it's necessary to restrict the scope of the planner to avoid too many calls for replanning and unnecessarily complex problems. This agent is set to consider only 3 parcels in a range of maximum 5 tiles from the agent and 2 delivery zones per parcel, while this might seem a very small number it's a sweet spot that allows for a good balance between speed and performance and encourages exploration, which in turn leads to better results.
- **BOOSTPDDL:** This agent is the same as the *ALLPDDL* agent but it uses the search boosting mechanism, the cache is filled with as many paths as possible in the first few seconds of the game, this allows to reduce the workload of the PDDL

solver. Additionally, its parcel sensing range is set to infinity like the BFS agent.

7.1 Benchmarking Scenarios

Agents are left to explore freely the map and collect parcels for 10 minutes, scores are recorded at the 5 minutes mark and at the end, results are averaged over multiple runs. Our scenarios try to mimic those seen during the live challenges done in class.

- **single-agent:** During the live challenges the map was filled with competitive agents moving at fast speed. While it's difficult to reproduce faithfully this scenario in a controlled environment, we have tried to simulate a similar situation by running said tests in a maps with a high density of parcels and randomly moving agents. For this reason the maps used for the benchmarking are **challenge_21**, which represents the simplest scenario of an open field map, **challenge_22**, which is more complex with narrower corridors that require a more careful planning and **challenge_23** which contains single-tile corridors that result in a large amount of blocked paths and replanning. Along our agent there are 5 other agents that move randomly and parcels are set to decay at a rate of 1s.
- **multi-agent:** For the multi-agent scenario we have decided to opt for slightly bigger maps and maps with more complex structures. The first map is **24c1_3** since it has a similar structure to challenge_22 but it's bigger and allows for more agents to move around, the second map is **24c1_5**, another map with narrow corridors but on a larger scale. All these maps are populated with 7 agents and parcels are set to decay at a rate of 1s.

7.2 Benchmarking Results

The results of the benchmarking tests are summarized in the following tables. The tables show the average **points per minute (PPM)** achieved by each agent in the different scenarios. The results are averaged over multiple runs to ensure consistency and reliability.

scenario	time	ALLPDDL	BOOSTPDDL	ALLBFS
challenge_21	*5*	341	566	681
challenge_21	*10*	351	518	680
challenge_22	*5*	378	466	657
challenge_22	*10*	401	523	670
challenge_23	*5*	272	410	513
challenge_23	*10*	377	394	530

Table 1. *Results of the single-agents benchmarks.*

All three configurations performed well in the single-agent settings, with the BFS agent achieving the highest PPM in all scenarios. The BFS agent's speed and efficiency in pathfinding allowed it to outperform the PDDL agents, especially in maps more crowded with more obstacles that tend to frequently trigger replanning. The ALLPDDL agent, while slower than the BFS agent, still managed to achieve competitive scores, showcasing the effectiveness of the PDDL solver in generating optimized plans. The BOOSTPDDL agent, with its search boosting mechanism, really benefited from the caching system, in particular in the first challenge where the chances of being blocked by other agents were lower and the cache hit rate was higher.

scenario	time	ALLPDDL	BOOSTPDDL	ALLBFS
24c1_3	*5*	246	208	545
24c1_3	*10*	246	209	544
24c1_5	*5*	258	419	721
24c1_5	*10*	287	413	737

Table 2. *Results of the multi-agents benchmarks.*

In the multi-agent scenario, the performance of the agents was more varied, with the BFS agent still achieving the highest PPM in most cases. It may look surprising how the scores in some scenarios are lower than the single-agent ones. However, this is only due to the fact that the maps play very differently: maps in the second bat are larger and much more sparse, therefore it takes more time to pick up and deliver parcels. On top of that, the cache is less effective as there's a larger variety of possible routes. Due to the differences in the first and second part it's incorrect to conclude that single agent perform better than their multi-agent counterpart, in fact the opposite is true. To further prove this point we ran some additional experiments. Table 3 shows a rundown of the performance of a fleet of 2 **ALLBFS** agents playing the dense scenarios of the single-agents benchmarks, the scores are almost doubled, which is the expected upper bound for the performance in this multi-agent system scenario. That being said, in the larger and sparser maps the pure PDDL agents struggled with their narrower field of vision and slower replannign times.

scenario	time	ALLBFS
challenge_21	*5*	1248
challenge_21	*10*	1251
challenge_22	*5*	1220
challenge_22	*10*	1226
challenge_23	*5*	1063
challenge_23	*10*	1088

Table 3. *For comparison, a fleet of two collaborative agents playing the single-agent scenarios.*

A noticeable trend across all scenarios shows how, as the game progresses, scores tend to increase. This is thanks to the *plansCache* that saves important milliseconds every time a path is computed, this allows for more replanning and more efficient use the PDDL solver.

7.3 Benchmarking Conslusions

It's clear that an important bottleneck in our implementation is the delay introduced by the external PDDL solver, and this is particularly evident in unpredictable scenarios where the cache is of little help. However, a reason why this gap is made evident is the exceptionally good performance of the planner that can really shine when enough compute power is available. The agents performed just as well in the single-agent scenario as in the multi-agent one and in all maps, the maximization-driven nature of the planner allows adapting to any kind of scenario and to find the best possible solution.

8. Challenges Faced and Solutions

We have encountered many challenges, mainly related to the limitations of the PDDL solver and their consequences. However, we have found workarounds for most of these. For more complex issues, we had to adopt entire new strategies and refactor parts of our code.

- **Local Planner Overhead:** The local PDDL planner, while faster than the online solver, still incurred considerable computational overhead for each individual call. To mitigate this, we implemented a strategy of bundling multiple planning queries into larger, consolidated calls. This approach aimed to amortize the fixed overhead costs across multiple planning problems, improving overall efficiency.
- **Limitations in PDDL Goal Specification:** When parallelizing goals in PDDL, we encountered a significant limitation: the inability to specify optional goals. Consequently, in scenarios where multiple goals were bundled together, if even a single goal was determined to be unachievable, the entire planning call would fail. To address this, we implemented a fallback mechanism that activates when a batch contains an unachievable goal. However, this solution introduces additional computational overhead, as it necessitates re-planning for the achievable subset of goals, thus slowing down the overall process.
- **Large Response Payload Processing:** As a consequence of bundling multiple planning queries, the size of the response payloads increased substantially. Processing these large responses introduced significant computational and time overhead, partially offsetting the gains from reduced call frequency.
- **Performance Limitations of Local Solver:** Despite being notably faster than the online solver, the local PDDL solver still fell short of the performance requirements for real-time agent decision-making in our dynamic environment. The solver's speed, while improved, remained a bottleneck in achieving truly responsive agent behavior.
- **BDI Loop Synchronization Issues:** The extended planning times, particularly when using the online solver or processing large bundled requests, occasionally led to synchronization issues within the Belief-Desire-Intention (BDI) loop. This temporal misalignment between the planning phase and the rapidly changing environment state resulted in agents acting on outdated information, potentially leading to suboptimal or irrelevant actions.

9. Future Improvements and Ideas

- **Solver efficiency:** after having analysed the inner functioning of the solver, we came to the conclusion that the solver per se is quite fast and can definitely live up to other algorithms' performances. However, we have also noticed that both the online solver and its local counterpart run in a Docker container have too much of an overhead for them to be efficient and practical in a real world, fast-paced scenario. As a matter of fact, while the algorithm for a BFS lookup took an average of 3ms, the overhead for the HTML post request, etc. took more than 100ms. As soon as the complexity of the problem rises, this issue becomes more and more relevant. In order to build a better multi-agent system, we would need to use a significantly more efficient planner.
- **Robust e2e communication protocols:** the API defines a protocol with two primitives (shout() and say()) that can be used to manage communication among agents. However, none of these takes security into consideration. In order to achieve a more secure and snitch-proof communication, an end-to-end encryption strategy could be implemented. This way, it would be much harder – and most likely non convenient – for an external agent to spoof itself as a man in the middle and intercept communications between our agents, or, in even worse cases, forge fake ones.
- **Implement more game-specific strategies:** the standard and naive approach to multi-agent coordination in the Deliveroo.js game is to optimize for the highest score, while not caring too much about what other agents are doing in the map. This is mainly done through dynamic task allocation, or by splitting the map in chunks and let each agent work in its own area. However, there can be many other greedy strategies that also aim at minimizing other teams' scores. For instance, one could split the agents' fleet in two teams, where one tries to pickup and deliver parcels, while the other aims at obstructing enemy teams by blocking agents' movement, or standing still on delivery zones, etc. This is just an example, but increasingly complex strategies could be thought in order to improve the overall fitness of a fleet of agents.

10. Conclusions

The development of this autonomous delivery agent system has demonstrated the effectiveness of combining genetic algorithms with the BDI architecture and PDDL dynamic path planning in a multi-agent environment. The genetic algorithm approach for plan generation proved to be particularly powerful, allowing agents to optimize their routes considering multiple factors such as parcel rewards, decay rates, and delivery costs, moreover the direct link between plan construction and score maximization allows for great flexibility and scalability regardless of the environment and number of agents. The implementation of a caching system for path finding significantly improved computational efficiency, enabling faster decision-making in real-time scenarios. Additionally, the multi-agent coordination mechanisms, including collision avoidance and task allocation, showcased the system's ability to handle complex, dynamic environments effectively. The project's success in balancing plan quality with computation time highlights the potential of this approach for real-world applications in autonomous delivery systems.

However, the project also revealed certain limitations and areas for improvement. The high computational cost of frequent replanning, especially when using the PDDL solver, sometimes led to performance issues in busier, more constrained environments. This suggests that future iterations might benefit from more advanced planning strategies or improved heuristics to reduce the frequency of recalculations. The challenge of handling parcel decay and dynamic obstacle avoidance also highlighted the need for more sophisticated predictive models. Furthermore, while the PDDL-based problem solving showed promise, its integration and performance compared to traditional methods like BFS were not even close, indicating an area for further optimization and research. These insights provide valuable direction for future enhancements, potentially exploring some of the solutions sketched above, or via advanced multi-objective optimization algorithms to address the complex trade-offs inherent in autonomous delivery systems.

References

[1] Deliveroo.js api. `https://github.com/unitn-ASA/Deliveroo.js`. Accessed: May 2024.

[2] Planning domains editor. `https://editor.planning.domains/`. Accessed: August 2024.

[3] Planutils github repository. `https://github.com/AI-Planning/planutils/tree/main`. Accessed: August 2024.

[4] unitn-asa / pddl-client. `https://www.npmjs.com/package/@unitn-asa/pddl-client`. Accessed: August 2024.

[5] Guy Desaulniers, Jacques Desrosiers, Andreas Erdmann, Marius M. Solomon, and Francois Soumis. Vrp with pickup and delivery. In Paolo Toth and Daniele Vigo, editors, *The Vehicle Routing Problem*, chapter 9, pages 225–242. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2002.

[6] Shaza Hanif and Tom Holvoet. Chapter 5 - applying delegate mas patterns in designing solutions for dynamic pickup and delivery problems. In Rosaldo J.F. Rossetti and Ronghui Liu, editors, *Advances in Artificial Transportation Systems and Simulation*, pages 79–102. Academic Press, Boston, 2015.

[7] Irfan Younas, Rassul Ayani, Johan Schubert, and Hirad Asadi. Using genetic algorithms in effects-based planning. In *2013 IEEE International Conference on Systems, Man, and Cybernetics*, pages 438–443. IEEE, 2013.