Autonomous Software Agents project

Bozzo Francesco, Izzo Federico University of Trento
Trento, Italy
francesco.bozzo@studenti.unitn.it, federico.izzo@studenti.unitn.it

Index Terms—**Autonoumous Software Agents; Single-agent; Multi-agent; BDI; A Star; Heuristic**

I. Agent

In the field of Computer Science, the term "agent" refers to an individual situated within an environment and capable of autonomously and flexibly taking actions to achieve its design objectives. Unlike traditional algorithms, agents do not require explicit definition of every edge case. Instead, a well-defined agent possesses a reasoning component that enables it to make decisions even in unknown situations. The flexibility of an agent can be assessed along two primary dimensions:

- Reactive: This dimension measures the delay required for the agent to respond to changes in the environment.
- Proactive: This dimension gauges the agent's ability to take proactive action to maximize future goals.

Communication models between agents and their environments vary depending on the specific characteristics of the environment and the agent itself. Generally, an agent perceives observations from the environment through sensors and carries out actions on the environment using actuators.

Autonomy is a fundamental characteristic of an agent. The internal decision-making process of an agent, often referred to as its "brain," should be capable of handling decisions with or without collected information. Furthermore, it should be able to adapt and evolve in response to potential changes in requirements.

Agents can be designed to solve tasks or goals. Task-oriented agents focus on accomplishing smaller objectives that contribute to the achievement of a larger final goal. On the other hand, goal-oriented agents receive a specific goal and autonomously determine a list of tasks necessary to fulfill the assigned goal.

A. Multi-agent system

A multi-agent system, as implied by its name, refers to a collection of agents situated within the same environment. Interactions within such a system can be broadly categorized into two types: cooperative and competitive. Throughout this project, we will delve into both of these interaction modes at various stages of development.

In a competitive system, multiple agents act in opposition to one another, where the overarching goal can be divided into sub-goals focused on maximizing personal rewards while minimizing opponents' gains. Within this scenario, the objective function may be shared among enemy agents, and it is also plausible to have multiple functions where each agent interferes with enemies solely to achieve its own goals.

On the other hand, a cooperative system consists of numerous agents working together to maximize a shared reward. In such systems, each agent must possess the capability to cooperate, coordinate, and engage in negotiation to the greatest extent possible. Cooperative systems can be further classified into two main categories:

- Simple (reciprocal) cooperation: This form of cooperation occurs when the benefits derived from collaboration outweigh the costs associated with the actions taken. It is considered the simplest type of cooperation as it leads to increased fitness for both the helper and the helped parties.
- Altruistic cooperation: In this case, the cost incurred by the individuals or species offering assistance surpasses the advantages gained. This approach is often regarded as more challenging since it cannot be readily explained by a purely "genetic-centric" perspective.

An essential aspect to consider when implementing a cooperative system is the communication mechanism. It should prioritize speed, reliability, and minimize delays as much as possible.

B. Architecture

There are several architectural options available for constructing an agent with the ability to operate within a specific environment. For our purposes, we have chosen to adopt the architecture outlined in the following pseudocode.

Algorithm 1 Agent control loop

```
 1: procedure AgentControLoop
 2:     $B \leftarrow B_0$                                  ▷ Belief set initialization
 3:     $I \leftarrow I_0$                                  ▷ Intention set initialization
 4:     while true do
 5:         perceive $\rho$
 6:         $B \leftarrow$ update($B, \rho$)                ▷ Belief set update
 7:         $D \leftarrow$ options($B$)                     ▷ Desires computation
 8:         $I \leftarrow$ filters($B, D, I$)               ▷ Intention update
 9:         $\pi \leftarrow$ plan($B, I$)                   ▷ Plan computation
10:         while not (empty($\pi$) or succeeded($I, B$) or impossible($I, B$)) do
11:             $\alpha \leftarrow hd(\pi)$                 ▷ Get next action
12:             execute($\alpha$)                           ▷ Execute action
13:             $\pi \leftarrow$ tail($\pi$)                ▷ Remove executed action
14:             perceive $\rho$
15:             $B \leftarrow$ update($B, \rho$)            ▷ Belief set update
16:             if reconsider(I, B) then
17:                 $D \leftarrow$ options($B$)             ▷ Desires computation
18:                 $I \leftarrow$ filters($B, D, I$)       ▷ Intention update
19:             end if
20:             if not sound($\pi, I, B$) then
21:                 $\pi \leftarrow$ plan($B, I$)           ▷ Plan computation
22:             end if
23:         end while
24:     end while
25: end procedure
```

After each communication with the environment, the belief set, denoted as B, undergoes an update. Subsequently, the desires set, referred to as D, is computed based on the updated belief set and then filtered to generate the intentions set, denoted as I. The intentions set, in conjunction with the belief set, is utilized to formulate a comprehensive plan, denoted as π, which aims to fulfill all the intentions. This plan is then executed incrementally, action by action, represented as α. Whenever the agent receives new information from the environment, the belief set, intentions set, and desires set are updated accordingly. Following this update, the agent can decide whether to generate a new plan or adhere to the existing one.

II. Deliveroo

The Deliveroo environment serves as the designated playground for developing our BDI agents. It encompasses a delivery-based game, where agents navigate on a two-dimensional grid. The objective of the game is to collect parcels dispersed throughout the map and swiftly deliver them to designated delivery zones. Each parcel possesses an assigned reward value, which may decay over time, and is granted to the agent responsible for its successful delivery.

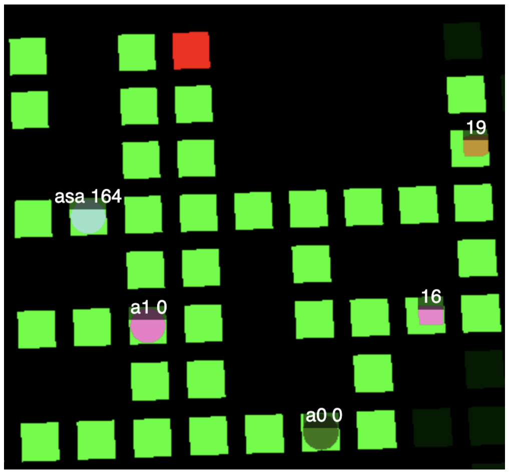

Figure 1. An example of a Deliveroo map.

The game can be played with either a single agent or multiple agents. In the latter case, agents are solid entities capable of obstructing each other's movement on the road.

Agents typically have limited perception of the world, restricting their awareness to other agents and parcels within a specified radius. Additionally, agents have the ability to carry and deliver multiple parcels simultaneously.

Deliveroo has been designed to offer diverse game scenarios through the manipulation of various parameters, including:

- Parcel: Generation interval, reward decay, and reward value distribution.
- Player: Number of steps and movement speed.
- External agent: Quantity and movement speed.
- Sensing: Radius of parcel and agent perception.
- Map: Definition of the grid board and classification of tiles as walkable, non-walkable, or delivery zones.
- Game clock.

Specifically, this examination project comprises two distinct deliverables, each with a different solution strategy:

- Single-agent: The agent operates independently, aiming to maximize its individual score.
- Multi-agent: The agent engages in communication with other agents to exchange information and collectively optimize the overall score of the group.

III. Single agent implementation

Considering the intricate nature of the environment, a multitude of tailored solutions have been implemented to address various complexities, difficulties, and challenges encountered.

A. Manhattan distance

The Manhattan distance is a mathematical function that offers a rapid estimation of the distance between two points. It is calculated by summing the absolute differences between the respective x-coordinates and y-coordinates of the two points:

$$d_{ab} = |b_x - a_x| + |b_y - a_y|$$

While this function may not be fully suitable for the Deliveroo game due to the presence of non-walkable tiles on the maps, it serves as a reasonable and computationally efficient heuristic for approximating distances in certain cases.

B. Problem parameters estimation

As detailed in Section II, various parameters can be adjusted during the initialization of the game. While certain parameters are directly communicated to the agent, others remain obscure and can only be estimated. To enhance the accuracy of estimating player rewards, we have implemented a learning mechanism aimed at approximating the player's speed and the decay of parcel rewards over time. This mechanism enables us to build a more precise estimation for optimizing player rewards.

1) Player speed: During the agent's movement within the map, it maintains a record of its position over time. This historical positional data, along with corresponding timestamps, allows the agent to estimate its own speed using the following algorithm:

Algorithm 2 Player speed estimation

```
 1: procedure UPDATEMAINPLAYERSPEEDESTIMATION($\mathcal{D}, s, \phi$)
 2:     deltas ← []                        ▷ Initialize an empty array of deltas
 3:     for each timestamp $t_i \in \mathcal{D}$ do
 4:         $deltas.append(t_i - t_{i-1})$  ▷ Delta of two consecutive timestamps
 5:     end for
 6:     $c \leftarrow s * (1 - \phi)$            ▷ Current speed contribution
 7:     $n \leftarrow$ avg$(deltas) * \phi$       ▷ New speed contribution
 8:     $s \leftarrow c + n$                    ▷ New estimation
 9:     return $c$
10: end procedure
```

Here, ϕ represents the learning rate hyper-parameter, which can be utilized to control the influence of the most recent measured instantaneous speed in relation to its historical value.

2) Parcel decay: Apart from estimating the agent's speed, our agents are also capable of estimating the decay of parcel rewards over time. Similar to the player speed estimation, the parcel decay is calculated based on differences in timestamps. Each timestamp is associated with a sensed reward update of a visible parcel, allowing us to estimate the decay of the parcel rewards as time progresses.

Algorithm 3 Get parcel decay estimation

```
1: procedure GETPARCELDECAYESTIMATION($\mathcal{D}$)
2:     deltas ← []                 ▷ Initialize an empty array of deltas
3:     for each timestamp $t_i \in \mathcal{D}$ do
4:         $deltas.append(t_i - t_{i-1})$  ▷ Delta of two consecutive timestamps
5:     end for
6:     return deltas
7: end procedure
```

Algorithm 4 Parcels decay estimation

```
1: procedure UPDATEPARCELSDECAYESTIMATION($\mathcal{P}, d, \phi_2$)
2:     deltas ← []                 ▷ Initialize an empty array of deltas
3:     for each parcel $p_i \in \mathcal{P}$ do
4:         $deltas.concat(getParcelDecayEstimation(p.timestamps))$ ▷
5:     end for
6:     $c \leftarrow d * (1 - \phi_2)$          ▷ Current decay contribution
7:     $n \leftarrow \text{avg}(deltas) * \phi_2$     ▷ New decay contribution
8:     $d \leftarrow c + n$                ▷ New estimation
9:     return $d$
10: end procedure
```

Here, ϕ_2 represents the learning rate, which controls the contribution of past estimations relative to the current estimated parcel decay. It allows us to regulate the influence of previous estimations when updating and refining the estimation of the parcel decay over time.

C. Probabilisitic model

Within the environment, multiple competitive agents coexist, and their effectiveness in picking up parcels significantly impacts the value of each individual parcel. To address this, we have developed a penalty value based on a probabilistic model that takes into account the potential plans of other competing agents.

The underlying concept behind this probabilistic model can be summarized as follows: "If there is a parcel available and I am the closest agent to it, I have a higher probability of reaching and acquiring it faster than any other agents. Consequently, this parcel should be given more weight and consideration, even if its assigned value is lower than that of other parcels located further away." This assertion can be formulated more formally as follows:

$$\text{penalty probability} = \frac{\sum_{a \in \mathcal{A}} \frac{d_{max} - d_{pa}}{d_{max}}}{|A|}$$

Here, we denote $\mathcal{A}$ as the set of opponent agents, d_{max} as the maximum distance between the parcel and the collective group comprising opponent agents, the main player, and cooperative agents. Additionally, d_{pa} represents the distance between the parcel and an opponent agent.

D. Potential parcel score

The process of parcel selection plays a crucial role in defining an effective agent. To accurately estimate the potential reward gain of a parcel, our agents consider various elements and metrics. Formally, the final reward for a parcel is computed as follows:

$$r_f = r - \left(d_{ap} * \frac{s_a}{decay} \right) - \left(d_{min} * \frac{s_a}{decay} \right) - r * \text{penalty probability}$$

Here, d_{ap} represents the distance between the agent and the parcel, s_a denotes the estimated speed of the agent, d_{min} represents the minimum distance between the parcel and the nearest delivery zone, and penalty probability corresponds to the probability calculated using the probabilistic model discussed in Section III-C.

The resulting formula takes into consideration multiple factors, including:

- The reward that the agent expends to approach the parcel and deliver it to the nearest delivery zone, which represents the minimum cost associated with delivering that particular parcel.
- An approximate estimation of other agents' intentions based on their distances from the parcels, incorporating a probabilistic model.

E. Distances cache

In order to optimize computational efficiency and enhance the accuracy of reward estimation, a cache is maintained to store distances between tiles throughout the map. Whenever a plan is generated, the distance between the starting point and any other tile along the path is stored in the cache. However, it should be noted that this approach does not guarantee the shortest route between two tiles. As a result, cache entries are updated whenever a smaller distance value is discovered. This caching mechanism acts as a form of learning, gradually improving over time.

Throughout the codebase, the cache is utilized in numerous instances. In the event of a cache miss, the agent resorts to using the Manhattan distance as a fallback measure. By employing this caching strategy, the goal is to strike a balance between computation efficiency and accurate reward estimation.

F. Replan

As Deliveroo is a dynamic game that involves simultaneous actions from multiple agents, we have implemented a mechanism to replan the actions of the current agent if it fails to execute a move within a specific time frame. This functionality allows agents to prevent getting stuck in narrow or crowded areas of the map. By triggering a replanning process when necessary, agents can adapt their actions and navigate through challenging situations more effectively.

IV. Multi agent implementation

The communication protocol is based on the provided library that offers multiple endpoints to handle different messages: say, shout, ask, and broadcast.

A. Information sharing agents

During the game, teams can share the data they sense from the environment between each other. For simplicity, for this specific purpose our agents use broadcast messages that are visible to every agent that is connected to the game.

In this way, every agent can build its one plan based on the information which is sensed by all the other agent across the map.

Moreover, agents also communicate by broadcast message the packages that they are going to pickup with the current plan they just generated. This mechanism helps in reducing the number of collisions of agents in the map and avoiding useless longer paths to pickup already taken parcels.

B. Leader-members agents

The leader negotiation is a fundamental step in the multi-agent architecture where one agent is chosen to be the leader who computes all the plans for the other agents.

1) Leader negotiation: The election of the leader is a well known problem in the computer science literature, nevertheless we have decided to keep a simple solution since the focus on the project was something else. The negotiation is achieved thanks to two message types: askforaleader and leader. The former is sent by every agent as a broadcast communication when they connect to the game (when they receive for the first time their position): as the name says, the message simply asks if a leader has already been elected. The latter is used to communicate the leader identity and it is used only by the agent who has been elected as the leader.

After the askforleader message is sent, a timeout of $2.5s$ is set. If no answer by an actual leader is received during that time interval, it means that the game has no already elected leader and consequentially the agent who asked for the leader identity will be elected as the leader itself. At this point, this agent communicates as a broadcast message that it has just elected itself as the leader.

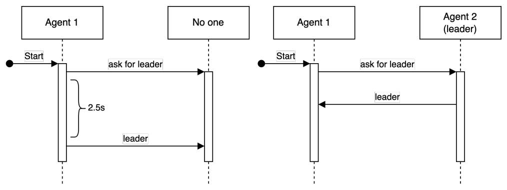

Figure 2. Leader negotation.

Please, consider that with this simple implementation very rare scenarios where two (or more) agents exactly connect at the same time could generate a race condition that would likely create more than one active leader.

V. Benchmarking

VI. Conclusion

VII. References