Autonomous Software Agents project

Bozzo Francesco, Izzo Federico

2023-07-11

Contents

1 Agent **3**
1.1 Multi-agent system . . . 3
1.2 Architecture . . . 3

2 Deliveroo **4**

3 Single agent implementation **5**
3.1 Manhattan distance . . . 5
3.2 Problem parameters estimation . . . 6
3.2.1 Player speed . . . 6
3.2.2 Parcel decay . . . 6
3.3 Probabilisitic model . . . 7
3.4 Potential parcel score . . . 7
3.5 Distances cache . . . 8
3.6 Replan . . . 8

4 Multi agent implementation **8**
4.1 Information sharing agents . . . 8
4.2 Leader-members agents . . . 8
4.2.1 Leader negotiation . . . 9

5 Benchmarking **9**

6 Conclusion **9**

7 References **10**

1 Agent

In the field of *Computer Science* an *agent* is an individual situated in some environment, and capable of flexible autonoumous action in that environment in order to meet its design objectives. With respect to a more traditional algorithm, there is no need of defining every edge case, a well defined agent should have a reasoning component capable of taking decisions even in unkown situations. The flexibility of an agent can be quantified across two main scales:

- reactive: delay required to respond to a change in the environment;
- proactive: ability of taking action in advance for the maximization of future goals.

Based on the environment and the agent itself there are different communication models but generally the agent perceives observations from the environment using sensors and performs actions against the environment employing actuators.

Another fundamental characteristic of an agent is its autonomy, the internal "brain" should handle the decision process with or without collected information, it should also evolve with respect to possible requirement changes.

Finally an agent can be designed to solve tasks or goals. When we talk about tasks we mean small objectives that must be completed to achieve the final bigger goal. On the other hand we have a goal agent, it takes the goal and it has to define autonomously a list of tasks to fulfill the assigned goal.

1.1 Multi-agent system

A multi-agent system, as suggested by the name, is a group of agents placed in same environment. There are two main type of interactions: *cooperative* and *competitive*. During this project we will focus on both of them in several stages of the development.

A competitive system is composed of many agents acting against each other, the goal can be divided into to sub-goals, maximizing the personal reward and minimize the oppenents' ones.

A cooperative system is composed of many agents acting to maximize the shared reward, each agent must be capable of cooperate, coordinate and negotiate as much as possible.

In a cooperative system an important implementation detail is the communication, it should be fast and reliable with the lowest possible delay.

1.2 Architecture

There exist multiple architectures that can be used to build an agent capable of acting in a given environment. We have decided to opt for the architecture described in the following pseudocode.

Algorithm 1 Agent control loop

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

A belief set B is updated after every environment communication, and it is then used to compute the desires set D which is filterd to create the intentions set I. Finally, the intentions set is used in combination with the belief set to create a plan π to satisfy all the intentions. The plan is then execute action by action α. Once the agent senses new information from the environment, the belief set, the intentions set, and the desires set are updated. After this update, the agent can decide whether to generate a new plan or stick with the current one.

2 Deliveroo

Deliveroo is the playground environement which has been provided to develop our BDI agents. It consists of a delivery-based game, where agents move on a bidimensional grid. The objective of the game is to pick up parcels that are scattered across the map and deliver them as fast as possible in the tile classified as delivery zones. Each parcel has an assigned reward value, which can decay over time, that is assigned to the agent that delivers it.

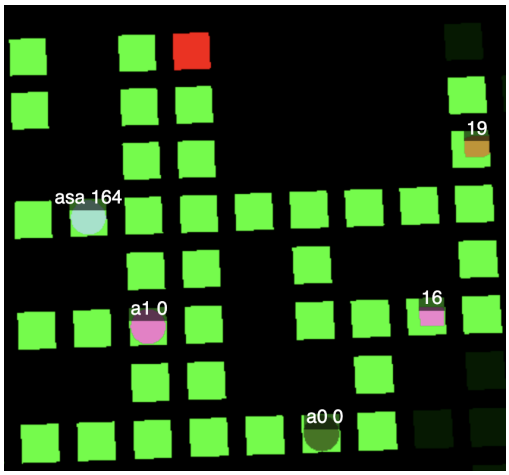

Figure 1: An example of a Deliveroo map.

The game can be played by either a single agent or multiple agents too. In the latter case, agents are solid entities and can block the road to each other.

Agents usually don't have the full perception of the world, but can perceive only other agents and parcels within a limited distance radius. Agents can also carry and deliver multiples parcels at the same time.

Deliveroo has been designed to offer many different game scenarios, thanks to the usage of many parameters: - *parcel*: generation interval, reward decay, reward value distribution; - *player*: steps number and movement speed; - *external agent*: quantity and movement speeed; - *sensing*: parcel and agent sensing radius; - *map*: definition of the grid board and tile classification (walkable, non-walkable, delivery) - *game clock*.

Specifically, this exam project is split in two delivebles, each one with a different solution strategy: - *single-agent*: the agent acts alone trying to maximize its score; - *multi-agent*: the agent communicates with others to share information and to maximize the overal score of the group.

3 Single agent implementation

Given the complexity of the environment, several ad-hoc solutions have been implemented to overcome different difficulties and challenges.

3.1 Manhattan distance

The *Manhattan distance* is function that can provide a quick estimation of the distance between two points.

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

Even though this function might be insufficient in the Deliveroo game since maps can have non-walkable tiles, it is still a decent and fast to compute heuristic that can be used to approximate distances.

3.2 Problem parameters estimation

As explained in Section 2, multiple parameters can be modified during the game initialization. While some of them are explicitly communicated to the agent, others are obscure and can be only estimated. For this reason, in order to build a more precise player reward estimation, we developed a learning meachnism to approximate the player speed and parcel reward decay over time.

3.2.1 Player speed

When our agent moves in the map, it is able to track its position over time: in this way, by using the historical positions alonside with timestamps the agent can estimate its own speed by using the following algorithm:

Algorithm 2 Player speed estimation

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

Where ϕ is the learning rate hyper-parameter that can be used to regulate the impact of the last measured instant speed with respect to its historical value.

3.2.2 Parcel decay

In addition to the agent speed estiamtion, our agents are able to estimate the parcel reward decay over time. Similarly to the player speed estimation, the parcel decay is computed from timestamp differences, where each timestamp is associated to a sensed reward update of a visible parcel.

Algorithm 3 Get parcel decay estimation

procedure GETPARCELDECAYESTIMATION($\mathcal{D}$)
  deltas $\leftarrow []$ ▷ Initialize an empty array of deltas
  for each timestamp $t_i \in \mathcal{D}$ **do**
    $deltas.append(t_i - t_{i-1})$ ▷ Delta of two consecutive timestamps
  end for
  return deltas
end procedure

Algorithm 4 Parcels decay estimation

procedure UPDATEPARCELSDECAYESTIMATION($\mathcal{P}, d, \phi_2$)
  deltas $\leftarrow []$ ▷ Initialize an empty array of deltas
  for each parcel $p_i \in \mathcal{P}$ **do**
    $deltas.concat(getParcelDecayEstimation(p.timestamps))$ ▷
  end for
  $c \leftarrow d * (1 - \phi_2)$ ▷ Current decay contribution
  $n \leftarrow$ avg$(deltas) * \phi_2$ ▷ New decay contribution
  $d \leftarrow c + n$ ▷ New estimation
  return d
end procedure

Where ϕ_2 is the learning rate and can be used to regulate the contribution of the past estimations with respect to the current estimated parcel decay.

3.3 Probabilisitic model

In the environment there may be multiple competitive agents, and their ability of picking up parcels highly influences the value of a parcel. For this reason, we have devised a penalty value based on a probabilistic model capable of taking into consideration the possible opponents' plans.

The main idea behind the probabilistic model is the following assertion: "if there is a parcel and I am the closest agent I can reach it faster than any other agents, consequentially that parcel should be taken more into consideration, even if its value is lower than other further parcels". More formally:

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

with $\mathcal{A}$ the set of opponent agents, d_{max} the maximum distance betweent the parcel and the union between oppoents agents, main player, and cooperative agents, d_{pa} the distance parcel oppenent agent.

3.4 Potential parcel score

The decision process behind the parcel selection is one of the key elements when defining a good agent. Many elements and metrics have been taken into consideration to better estimate the potential reward gain of a parcel. More formally, out agents compute the final reward as:

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

where d_{ap} is the distance between the agent and the parcel, s_a is the estimatated speed, d_{min} is the minimum distance between the parcel and the closest delivery zone, and penalty probability is the probability computed on Section 3.3.

The resulting formula takes into account multiple factors: - the agent reward that is spent to approach the parcel and delivery it to the closest delivery zone (minimum cost to deliver that package) - rough estimation of other agents' intentions through their distance from parcels by using a probabilistic model

3.5 Distances cache

To save computation power and to provide a more precise reward estimation, a cache is mantanied to store distances between tiles across the map. Every time a plan is generated, the distance between the starting point and any other tile in the path is stored in the cache. Since this approach does not guarantee that the generated path is the shorted route between two tiles, cache entries are updated once a smaller value is found. Therefore the caching approach is meant to improve over time as a sort of learning mechanism. In general, the cache is used many times in the codebase: in case of a cache miss, the agent uses the Manhattan distance as fallback.

3.6 Replan

Since Deliveroo is a dynamic game with potentially multiple agents acting at the same time, we implemented a mechanism replan the current agent's actions when it fails to perform a move for a specific amount of time. This functionality enables agents to avoid them from stucking in narrow and crowded areas of the map.

4 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.

4.1 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.

4.2 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.

4.2.1 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.

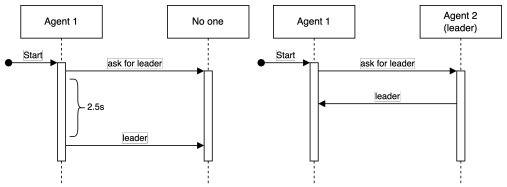

Figure 2: Leader negotation.

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.

5 Benchmarking

6 Conclusion

7 References