Autonomous Software Agents - Project Report

Silvano Vento Maddonni - 247370
silvano.maddonni@studenti.unitn.it

Introduction

This project involves the development of an autonomous software agent that is designed to play in the Deliveroo environment. The agent (composed of two agents for part 2) was designed to perform a variety of tasks autonomously, adapting to different scenarios and things happening around him.

Project folder description

`ag1.js` is the single agent, `ag2_1.js` and `ag2_2.js` are the two multi-agents, but they are tied together by the `ag3.js` script that is the one running. The last two files are the `domain-d.ppdl`, used in the PDDL planner, and a file with some functions used by the agents. Some of the agents' behavior can be changed with some flag variables at the beginning of the files (explained later and with comments in the code).

```
const when_idle = 1;          const multi_agents = 1;
var split_map = 1;            const use_pddl = 1;
```

Part 1 - Single Agent

For the first part, a single agent has to be implemented. The single agent `ag1.js` (named SVM in-game) built for challenge1 was gradually improved adding new things and later split in two similar copies, so the Agent for part1 is actually the same that operates in the multi-agent setup, but with a flag variable `multi_agents = 0` that changes many interactions.

The architecture of the agent is based on BDI with IntentionRevision. In order to reach goals, a plan is selected from a Plan Library that represents the agent's procedural knowledge. When sensing the environment around the agent, new intentions are going to be generated and a plan from the Plan Library is selected if it is applicable.

Sensing

In order to build an internal representation of the environment surrounding the agent, many sensing functions are used, starting with `onConfig`, triggered when connected. `onYou` and `onMap` are used to generate knowledge of the map and the position and information of the agent. `onAgentSensing` handles other agents and `onParcelSensing` regulates the main action of the agent: picking up (and later delivering) the parcels.

When a parcel is spotted, if it is not already carried by someone and it is the best option selected, the intention *go_pick_up* will be pushed.

If no parcels are available to being picked up and the agent is already carrying some parcels, it will go delivery, otherwise the idle behavior is called.

The idle behavior is regulated by the `when_idle` flag and it will call the plan explore random or the plan standing still.

Plan Library

Inside the Intention class, there is a for-loop that checks all the plans in the PlanLibrary and will try to achieve the current intention with one of the applicable plans. Each plan has a condition to be applicable that is represented by the name of the intention. Here the main plans are presented.

go_to (x,y) - BlindMove and PDDL_move

At the core of all other plans. Every time the agent needs to go somewhere, the *go_to* sub-intention is called. Depending on whether the `use_pddl` flag is set to 0 or 1, BlindMove or PDDL_Move will be used.

In BlindMove, the agent will try to get closer to (x,y) by performing a step by step in the x and y directions and updates its position after each move. If both moves fail, a simple backtracking mechanism is activated that will make the agent perform random moves to escape the stuck situation and then continues towards the target.

With PDDL_move, a call to the PDDLpathPlanner will be made and it will return the solution as a set of operations (steps) that will be then executed by the executors (`client.moves`). For example:

```
Plan found:
- RIGHT,ID_BB5BB5A2B07,T1_7,T2_7
- RIGHT,ID_BB5BB5A2B07,T2_7,T3_7
- UP,ID_BB5BB5A2B07,T3_7,T3_8
- UP,ID_BB5BB5A2B07,T3_8,T3_9
- LEFT,ID_BB5BB5A2B07,T3_9,T3_8
```

GoPickup and GoDeliver - (x,y)

GoPickup and GoDeliver are two simple plans that, when used, they will call a *go_to* (x,y) and then after that perform, respectively, `client.pickup()` or `client.putdown()`. The target (x,y) passed represents the location of the parcel to be picked up or the coordinates of the delivery tile.

Examples in the code:

```
options.push( [ 'go_pick_up', parcel.x, parcel.y, parcel.id]);
...
myAgent.push( ['deliv1', nearestDelivery(me.x,me.y).x, nearestDelivery(me.x,me.y).y])
```

ExplRandom

If there is at least one, it will pick one random spawnable tile, otherwise it will pick a random tile, and move the agent there. The condition to be spawnable was added later. Before that, it could move in random areas that wouldn't make sense to reach.

If the function is called while on a spawnable tile, another random spawnable tile will be selected, unless that tile is the only spawnable. In that case, once reached, since it is a spawnable tile, instead of just standing on it thus preventing the parcel to spawn, a camping-cross action is performed, in order to keep moving around the tile while waiting for a parcel to spawn.

RandomMove and ExplFar3

RandomMove will pick one random direction and try to move there. If failed (blocked), it will try all the others. If none worked, it will wait 1ms and retry, as likely blocked by another agent. ExplFar3 is a plan that will move the agent to the farthest corner of the map from his position. Currently not used.

PPDL and PathPlanning

A PDDL planner was implemented and used as a Path-Finder. A class called PathPlanning handles this. Inside it, a constructor is called the first time, initializing the map and defining the initial problem.

The domain is read from the `domain-d.pddl` file, in which are defined the predicates and the four possible actions. The problem is dynamically changed according to the environment, and it is updated using the `updateTarget(agentID, startX, startY, targetX, targetY)` function.

The PDDLpathPlanner will return the solution as a set of operations (steps) that will be then executed by the executors. For example:

```
Plan found:
- RIGHT,ID_BB5BB5A2B07,T1_7,T2_7
- RIGHT,ID_BB5BB5A2B07,T2_7,T3_7
- UP,ID_BB5BB5A2B07,T3_7,T3_8
- UP,ID_BB5BB5A2B07,T3_8,T3_9
- LEFT,ID_BB5BB5A2B07,T3_9,T3_8
```

The executors are defined as the `client.moves()`.

```
const executors = [
    { name: 'left', executor: () => client.move('left')},
    { name: 'right', executor: () => client.move('right')},
    { name: 'up', executor: () => client.move('up')},
    { name: 'down', executor: () => client.move('down')},
];
```

initializeMap()

The `initializeMap` function is used to create the objects and initial state of the PDDL problem. It gets called only once, since the map stays the same during the game. It first recreates internally the map from the tiles received from the server. After that, it 'converts' it to PDDL objects and initial state. Directional relationships are also calculated here, as the movement actions in the domain file require a full mapping of every relationship between tiles.

domain-d.pddl

```
(define (domain default)
    (:requirements :strips)
    (:predicates
        ...
        (right ?t1 ?t2)
        (left ?t1 ?t2)
        (up ?t1 ?t2)
        (down ?t1 ?t2)
        ...

    (:action right
        :parameters (?me ?from ?to)
        :precondition (and
            (me ?me)
            (at ?me ?from)
            (not (blocked ?to))
            (right ?from ?to)
        )
        :effect (and
            (at ?me ?to)
            (not (at ?me ?from))
        )
    )
    ...
```

Part 2 - Multi-Agents

The second part consists in building a team of two coordinated agents. The two agents are similar copies of the Agent used for part1, but with some modifications and added complexity because of coordination, exchange of information and different plans. The two agents are represented by `ag2_1.js` and `ag2_2.js` and a file called `ag3.js` is used to run them together using *spawn*. In the game are called SVM and SVM_2.

```
ag3.js
  ...
  runScript("ag2_1.js");
  runScript("ag2_2.js");
```

Two Agents

Many different strategies and approaches can be built with two agents. In this section are presented the characteristics that are present in the architecture that has been implemented. The two agents are not completely symmetric. While not being a master-slave structure, ag2_1 keeps being the main one. This is reflected in some choices and in information asking/exchanging. ag2_2 also has some special functions.

Map Split

The original map is split vertically in two parts, creating map1 and map2, assigned to ag1 and ag2. This map split doesn't preclude the agents to go to the other side: if for example ag2 has a parcel and delivery action is selected and the closest delivery tile is in map1, it will go there.

It is mainly used when selecting Expl_Random location to ensuring that they cover different areas. It also doesn't matter where they spawn.

Message Exchange - Communicating

Agent1 and Agent2 communicate using `client.onMSG` and `client.ask/client.say`. Both the agents are configured to reply to: *parcel_spotted*, *distance_to*, *go_to*. Agent2 has 2 additional onMSG: *distance_to_low1* and *isolated_section*. Inside the messages, also coordinates, distances and other info are passed. In many sections of the code, one agent asks the other something and then wait the reply and then his actions are decided based on the reply received.

```
client.onMsg( (id, name, msg, reply) => {

    if (msg.hello == 'parcel_spotted') {
        if (distance({x:msg.px,y:msg.py},{x:me.x,y:me.y}) <= msg.d) {
            reply(false);
        }
        else {
            reply(true);
        }
    }
    ...
```

Parcels

Parcel spotting and picking up is not limited to map1/map2, as this would not be beneficial. When spotting a new parcel, the agent that spotted it will send a message to the other, communicating the parcel position, id, and his distance from it. The other agent will calculate his distance from the parcel, and reply true or false based on that.

In this multi-agents scenario, each agent has also an `ignored_parcels` list: if a parcel is in inside it, it will be ignored for any purpose. Ignored parcels are used for `isolated_sections` (see later).

```
if (me.carrying.size<4 && !parcel.carriedBy && reply &&
    (!ignored_parcels.includes(parcel.id))) {

        options.push( [ 'go_pick_up', parcel.x, parcel.y, parcel.id]);
}
```

Isolated Sections and extra plans

Inside the initializeMap function, we also check for isolated sections and if the agent is inside one of them. An isolated sectionis a part of the map disconnected from the rest, like a 1-tile column. The check is performed by a function present in `functions2.js`. Both the agents check if there are any isolated sections and if they are inside one of them. If Agent1 finds out that he is inside an isolated section, an ask-message is sent to Agent2 asking if he also is in the same section. If they both are, the `isolated_section` plan is pushed. Otherwise, the `isolated_single` is pushed.

```
if (reply) {
    myAgent.push( ['isolated_section', isInIsolatedSection]);
} else {
isolated_single = 1; myAgent.push( ['isolated_section_single', isInIsolatedSection]);
}
```

isolated_single

```
Agent1: Is my position inside isolated sections? true
IntentionRevisionReplace.push [
  'isolated_section_single',
  [
    [ 16, 0 ],  [ 16, 1 ],
    [ 16, 2 ],  [ 16, 3 ],
    [ 16, 4 ],  [ 16, 5 ],
    [ 16, 6 ],  [ 16, 7 ],
    [ 16, 8 ],  [ 16, 9 ],
    [ 16, 10 ], [ 16, 11 ],
    [ 16, 12 ], [ 16, 13 ],
    [ 16, 14 ], [ 16, 15 ],
    [ 16, 16 ], [ 16, 17 ],
    [ 16, 18 ], [ 16, 19 ]
  ]
]
```

Inside `isolated_single`, the map1 of the agent1 gets updated by eliminating anything that is outside the `section_single`, as unreachable. Any tiles not anymore in map1 will be completely ignored, also for example for parcels.

```
if (map1.xy(parcel.x, parcel.y) == undefined){ignored_parcels.push(parcel.id)};
```

isolated_section

This is the case where both the agents are stucked inside the same isolated section. In case the situation is affine to a 1-tile column section with a delivery only on one side, a new strategy has to be implemented. As first thing, after realizing that the agents are in this situation, a new location (`base_x`, `base_y`) is assigned to each agent. It represents the closest endpoint of the `isolated_section` to each agent. So for example, for the previous `isolated_section` showed, those would be [16,0] for Ag1 (if he is the one 'on the left', vice versa otherwise) and [16,19] for Ag2. This is obtained by ag1 sending a msg to ag2 asking for his *distance_to_low1*.

After this, the GoDeliver plan in the PlanLibrary is substituted with GoDeliver2. Inside this new delivery plan, when an agent has a parcel that need to be delivered, it will calculate its distance to it and then ask the other agent if he's closer.

```
reply = await client.ask( '35d84ac07da', {
    hello: 'distance_to',
    x: x_r,
    y: y_r,
    my_d: distance({x:x_r,y:y_r},{x:me.x,y:me.y})
} );
```

If ag2 is not, ag1 will deliver (as he is the one on the side of delivery, in the isolated column). If the other agent is closer, then ag1 is not the one on the side of delivery, and so it will putdown the parcel, add it to the ignored list, move back (to his `base_x` and `base_y`) and tell ag2 to go pick up the parcel and deliver it.

```
for (let pid of me.carrying.keys()){
    ignored_parcels.push(pid);
    }

await client.putdown();
await this.subIntention( ['go_to', base_x, base_y] );

await client.say( '35d84ac07da', {
    hello: 'go_to',
    x: base_x,
    y: base.y,
} );
```

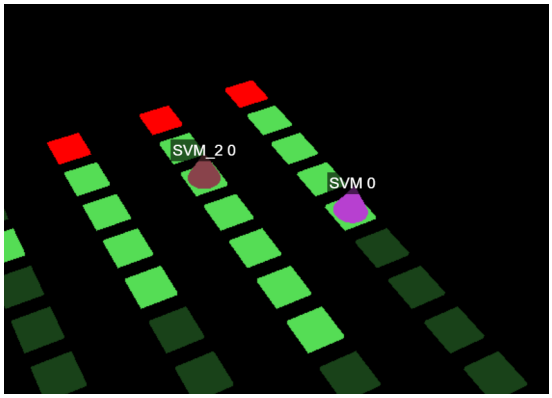

Figure 1: `isolated_single`

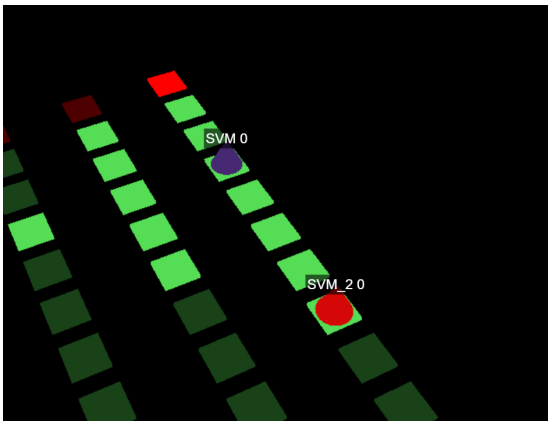

Figure 2: `isolated_section`

Agent2 special plans: BlockDelivery and BlockSpawn

Agent2 has two exclusive special plans: BlockDelivery and BlockSpawn. The idea is that, when the map has a configuration where doing that is beneficial, the Agent2 can play a disruptive role on a part of the map, while the Agent1 will play normally in his part.

The condition to apply **BlockDelivery** is:

```
if (map.deliv().length <= 5 && map2.deliv().length != 0 && map2.deliv().length <= 2)
```

If that is true, ag2 will select one random delivery tile in map2, go there and permanently stand on it. The idea is that we want a situation where there are not many delivery tiles (<=5) in the whole map (map= map1 + map2), at least one in map2 side but not too many (<=2) otherwise it wouldn't cause too much damage to enemy agents.

The condition to apply **BlockSpawn** is:

```
if (map.spwn().length <= 5 && map2.spwn().length != 0 && map2.spwn().length <= 2)
```

Similar behavior as the BlockDelivery. The agent will stand on the spawn tile blocking it.