Autonomous Software Agents
LogicLegion - Final Report

10 Giugno, 2024

github.com/giulianbiolo/asa_agent

Giulian Biolo - 248726 giulian.biolo@studenti.unitn.it

Project Overview & Objectives:

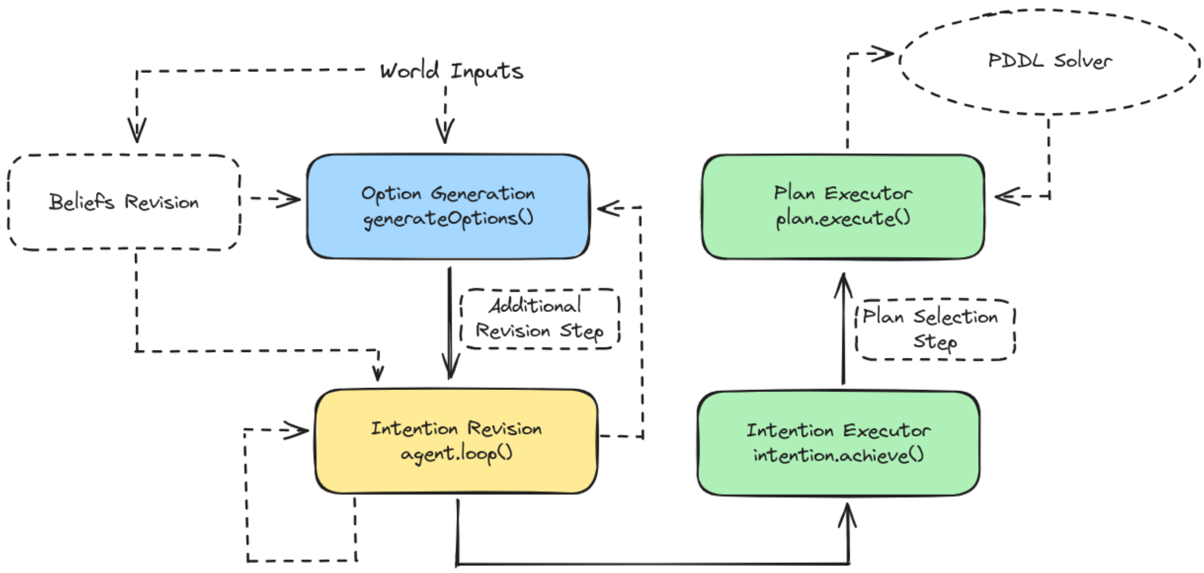
The following is a report regarding the detailed steps taken to implement and execute an autonomous agent capable of collecting packages and delivering them to the designated delivery zones in the Deliveroo.js simulation environment. It is divided in two main parts:

- The first part focuses on the implementation and testing of Agent A, which is responsible for achieving it's tasks autonomously.
- The second part focuses on the implementation and testing of the code necessary to coordinate two agents, with same underlying engine as the first part agent, such that they can work together to achieve the same goals.

Design and Implementation of Agent A:

Overview Of The Core Engine:

The basic model around which the entire codebase is designed is the BDI Model.
Every world sensing events executes a belief revision step, which updates the agent's beliefs about the world.
The Agent implements a loop that continously updates the actions it could perform (Options) and pushes the actions it decides to perform (Intentions) to the execution queue.
Every time a new event gets sensed by the agent, the method: `generateOptions()` is called.
The following is a diagram of the core engine's flow:



Sensing The Environment:

Parcels Sensing:

The "onParcelsSensing" hook is called every time a new parcel enters the field of view of the agent. For each we store the parcel's position and score in the agent's beliefs:

```
type Parcel = {
  id: string | null,
  position: Point2D,
  carriedBy: string | null,
  reward: number | null,
};
const parcels: Map<string, Parcel> = new Map();
```

We don't make the parcel decay with time as it's not particularly relevant. If the agent decides to pick it up, the hook will be called again each time the parcel updates it's score keeping it's belief up to date. Otherwise, if the agent decides to ignore it, having the parcel remain in the memory of the agent might turn advantageous in the future as an alternative to a sort of random walk when the agent runs out of options.

Tiles Sensing:

The "onTile" hook is called at the start for each tile in the map. We store the tiles on a map in the agent's beliefs, which will be used to plan the agent's path. Every delivery tile is stored separately in an array of delivery tiles, and the same goes for spawner tiles as follows:

```
type Point2D = { x: number, y: number };
var tiles: Array<Array<number>> = [];
var delivery_tiles: Array<Point2D> = [];
var spawner_tiles: Array<Point2D> = [];
```

Those will turn useful when the agent will have to plan it's path to deliver a parcel or to random walk, as random walking is more efficient if done over spawner tiles.

Agents Sensing:

the "onAgent" hook is called every time a new agent enters the field of view of the agent. We store the agent's position and id in the agent's beliefs. At the same time, other agents are considered as obstacles when planning the agent's path, and as such, the agent will usually try to avoid them, whenever possible. Agents are saved as follows:

```
type Agent = {
  id: string,
  name: string,
  position: Point2D,
  score: number,
  lastUpdate: number,
};
```

Also, agents older than 3 seconds are considered as not relevant anymore and are removed from the agent's beliefs, and their tiles are not considered as obstacles anymore, this way the agent doesn't get stuck with no path plans to follow due to old beliefs.

Path Planning:

In the first implementation the agent needed a fast and reliable path finding algorithm. Given the map size and the performance of javascript, BFS and DFS algorithms were discarded as they might have been too slow. Given the geometry of most of the maps, the more suitable algorithm between Djikstra and A* seemed to be A*, thus our choice feel on the A* algorithm.

In the second implementation of the path finding algorithm, we used PDDL. As such we wrote a parser for the map and the agent's beliefs, and we used the PDDL online solver api's to obtain the plan. The PDDL planner is called every time the agent has to plan a path, be it for randomly walking, going to pick up a parcel or delivering parcels. While for the simple actions of picking up and putting down parcels, if the planner doesn't include those actions in the plan, the agent will execute them anyway, as they are not particularly complex actions and the latency of the planner would make the agent miss the opportunity of gaining more points.

The BDI Model Implementation:

Belief Revision Loop:

Right after any of the above sensing functions is called, the agent will call the `generateOptions()` function, which will update the agent's beliefs and generate the possible actions the agent can perform. The following is the pseudocode of the `generateOptions()` function:

```
function generateOptions(): void {
  let options: Option[] = [];
  if (carryingParcels < CONFIG.MAX_CARRYING_PARCELS)
    evalGoPickUpParcels(options);
  if (carryingParcels > 0)
    evalGoDeliverParcels(options);
  if (options == [])
    evalRandomWalk(options);
  let bestOption = chooseBestOption(options);
  myAgent.push(bestOption); // Push the best option to the intentions
}
// The generateOptions() method is then called every MOVEMENT_DURATION milliseconds
setInterval(() => { if (pathFindInit) { generateOptions(); } }, CONFIG.MOVEMENT_DURATION);
```

The `chooseBestOption()` function will choose the best option based on the agent's beliefs and the current state of the world. To do this it needs to give a score to each option, in terms of how much it will benefit the agent. The function that implements this behaviour is the `realProfit()`, in the utils.ts module. The following is the main equation used to calculate the score of an option, which would be the `profit`:

$$d_{\text{multiplier}} = \texttt{MOVEMENT\_DURATION / PARCEL\_DECADING\_INTERVAL}$$
$$l_{\text{delivery}} = \text{carry}_{\text{parcels}} * d_{\text{delivery}} * d_{\text{multiplier}}$$
$$l_{\text{pickup}} = \text{carry}_{\text{parcels}} * d_{\text{parcel}} * d_{\text{multiplier}} + (\text{carry}_{\text{parcels}} + 1) + d_{\text{delivery\_from\_parcel}} * d_{\text{multiplier}}$$
$$g_{\text{pickup}} = p_{\text{reward}} - d_{\text{parcel}} * d_{\text{multiplier}}$$
$$\texttt{profit} = g_{\text{pickup}} - (l_{\text{pickup}} - l_{\text{delivery}})$$

Basically what this does is it estimates the profit of going to pick up a parcel and deliver it taking into consideration not only the score loss caused by the time taken to reach it, but also taking into consideration the time needed to reach the nearest delivery tile from there, and also taking into consideration the total loss of points of the parcels already carried by the agent.

Intention Revision Loop:

When the `generateOptions()` ends, it pushes the best option to the intentions by calling the `myAgent.push()` method. This method goes through various checks to decide wether it's better to keep achieving the current intention or to switch to the new one. The following is the pseudocode of the `push()` method:

```
async function push(newOption: Option): void {
  if (intentions.length == 0) {
    intentions.push(new Intention(newOption));
    return;
  }
  let currentIntention: Intention = intentions[intentions.length - 1];
  if (currentIntention.option === newOption) { return; }
  if (isRndWlk(currentIntention) && !isRndWlk(newOption)) {
    intentions.push(new Intention(newOption));
    return;
  }
  if (bothArePickUp(currentIntention, newOption)) { selectHighestProfitPickup(); }
  if (bothAreDeliver(currentIntention, newOption)) { selectHighestProfitDeliver(); }
```

```
  intentions.push(new Intention(newOption));
}
```

So basically, aside from the trivial situations, this policy prefers to do anything else rather that random walking, and it selects the highest scoring plan when both are of the same desire. Otherwise, it replaces the old intention with the new one.

At the same time, we have a `loop()` method executing continuously, which will await the execution of the various intentions, and at the same time it executes some basic integrity checks on the options being executed, and it will call the `generateOptions()` method when the agent has no intentions left to execute.

The following is the pseudocode of the `loop()` method:

```
async function loop(): void {
  while(true) {
    let currentIntention: Intention = intentions[intentions.length - 1];
    if (isGoPickUp(currentIntention)) { checkValidPickUp(); }
    if (isGoPutDown(currentIntention)) { checkValidPutDown(); }
    if (isRndWlk(currentIntention)) { generateOptions(); }
    try { await currentIntention.achieve(); }
    catch (e) { console.error(e); }
  }
}
```

Plan Implementation:

Plans are classes implementing the `PlanInterface` interface, which has two methods: `isApplicableTo()` and `execute()`. The `isApplicableTo()` method will return true if the plan is applicable for achieving the given intention's option, and the `execute()` method implements the actions that need to take place to achieve the given intention.

The various plans inherit from a parent class called `Plan` which implements the `subIntention()` method, used to create and achieve sub-intentions, and the `stop()` method, used to halt the execution of the current Plan, with all it's subIntentions, in case the intention revision loop has decided to replace the currentIntention. The following is a simplified representation of the code in the `plan.ts` module:

```
interface PlanInterface {
  isApplicableTo(option: Option): boolean;
  execute(option: Option): Promise<boolean>;
}
class Plan implements PlanInterface {
  isApplicableTo(option: Option) { return false; }
  async execute(option: Option): Promise<boolean> { return false; }
  public stopped: boolean = false;
  stop() {
    this.stopped = true;
    for (const i of this.#sub_intentions) { i.stop(); }
    this.stopped = false;
  }
  #sub_intentions: Array<Intention> = [];
  async subIntention(predicate: Option): Promise<boolean | Intention> {
    const sub_intention: Intention = new Intention(predicate);
    this.#sub_intentions.push(sub_intention);
    return await sub_intention.achieve();
  }
}
// PickUp Example, the same goes for all the other plans
class PickUp extends Plan implements PlanInterface {
  isApplicableTo(option: Option): boolean { return option.desire === Desire.PICK_UP; }
  async execute(option: Option): Promise<boolean> {
    if (this.stopped) { throw "stopped"; }
    let res = tryPickUp(option);
    if (!res) { throw "stucked"; }
    return true;
  }
}
```

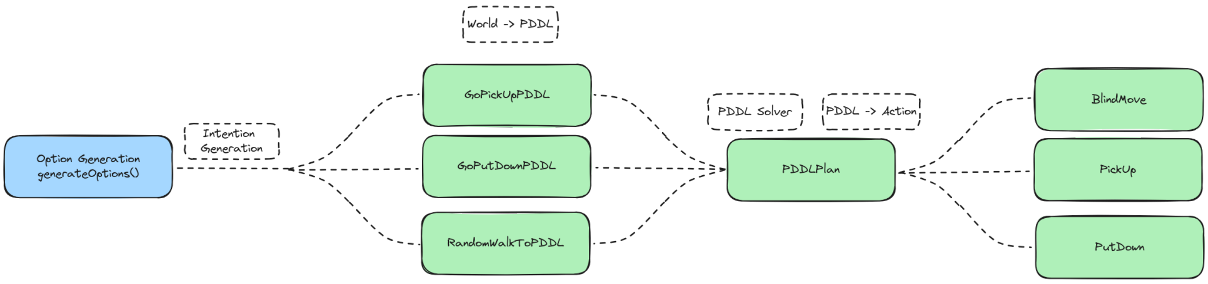
When using the basic A* planner, the following plan classes will be used:
- `GoPickUp`: used to pick up parcels
- `GoPutDown`: used to deliver parcels
- `AStarMove`: used to move to any tile with the A* planner

While the following are plan classes used both in the A* planner and in the PDDL planner:
- `PickUp`: used to execute the action of picking up a parcel
- `PutDown`: used to execute the action of delivering a parcel

PDDL-Based Planning:

To implement the PDDL-based planning, the following plans have been written:
- `GoPickUpPDDL`: used to pick up parcels
- `GoPutDownPDDL`: used to deliver parcels
- `RandomWalkToPDDL`: used to randomly walk
- `PDDLPlan`: used to go to a specific tile
- `BlindMove`: used to move to a specific tile

All these plan classes do not execute the actions by themselfes, instead they use the methods in the `pddl.ts` module to get the current representation of the world in PDDL format, then they add a goal to the `problem.pddl` representation, based on the option they are trying to achieve, and finally they call `PDDLPlan` as a subIntention. The `PDDLPlan` class will then call the PDDL solver APIs with the given problem and domain representations, and it will parse and execute the produced `plan.pddl` representation as various subIntentions of `BlindMove` and `PickUp` or `PutDown` plans. The following chart represents the flow of the PDDL-based planning:



As for the PDDL Planner in use, we tried using various solutions. One of which was the fast-downward planner, which can still be seen implemented in code in the `planner/PddlOnlineSolver.ts` module as the `offlineSolver()` function. At the moment it requires the user to install the fast-downward planner under the `/opt` folder of the system. This local planning solution is only in experimental state as it requires the user to install the planner on the system, and it still requires improvements. It is extremely faster than the online planner, but it achieves it's speed by using greedy algorithms to solve the problem, and as such it might not always find the optimal solution. Not only that but it also has a sizable impact on the performance of the machine on which it is running, as the agent will execute it possiblt even multiple times a second.

Given these limitations, we decided to stick to the online planner, which is slower but more reliable overall.