Autonomous software agents exam project by Francesco Rigoni
student id: 241188
email : francesco.rigoni-2@studenti.unitn.it

Single Agent

Data Structures

- map_beliefs: A Beliefset imported from pddl-client package, used to maintain the beliefs about the map that are supposed to be static separated from the ones concerning agents and parcels.
- me: simple data structure containing all the information about the agent such as position, carried parcels and id.
- parcels: A Map used to contain the knowledge on the parcels seen by the agent.
- agents: A Map used to contain the knowledge on the agents seen by the agent.
- planLibrary: An Array that contains the classes used to represent intentions.
- myAgent: An instance of the IntentionRevision class used to represent the agent.

Execution flow

- **Defined events**

    - onConfig: Same function as the one used in the benchmark agent, plus a setInterval that calls a function to update the beliefs of the agent every PARCEL_DECADING_INTERVAL

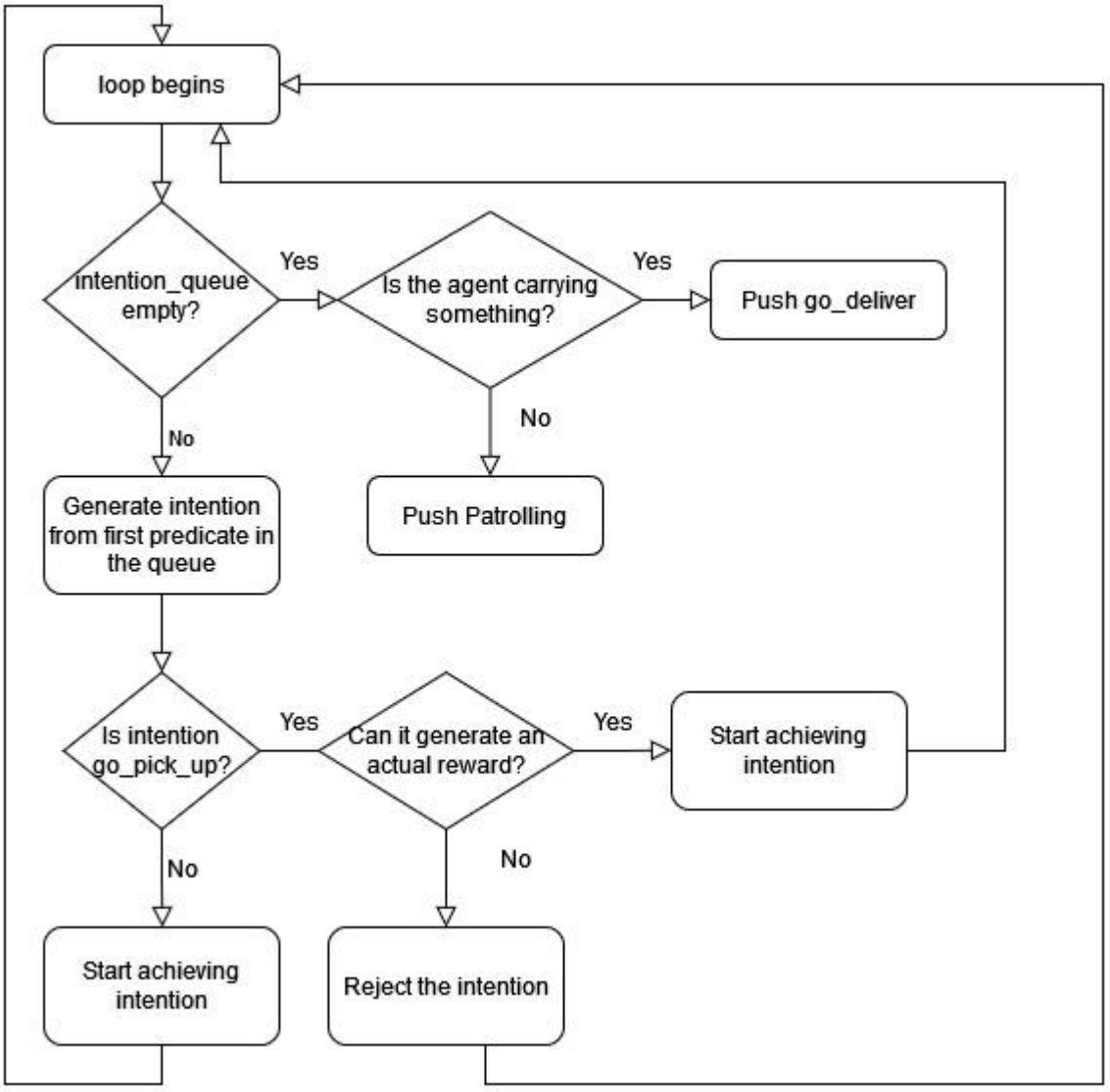
    - onMap: Used to generate the mapBeliefs
    - onParcelsSensing: Used to update the Parcels Map
    - onAgentsSensing: Used to update the agents Map
    - on("new_parcel"): Called by onParcelsSensing if a new parcel is seen,

```JavaScript
sensingEmitter.on("new_parcel", async () => {

	console.log('I saw a new parcel')
	console.log(me.carrying)
	myAgent.intention_queue = []
	let current_position = { ...me }
	let tmp = Array.from(parcels.values())
	tmp.sort((o1, o2) => distance({ x: o2.x, y: o2.y }, current_position) -
distance({ x: o1.x, y: o1.y }, current_position))
	for (const par of tmp) {
		let pred = ['go_pick_up', par.x, par.y, par.id, par.reward]
```

```
                if (par && !par.carriedBy) {
                        myAgent.push(pred)
                }
        }
})
```

- **Main loop**



- **Types of predicate**

- Go_deliver: Calculates nearest delivery tile, creates the subIntention go_to and, once it resolves, puts down the carried parcels and updates the corresponding data structures.
- Go_pickup: Creates the subIntention go_to and waits for it to resolve, picks up the parcel and updates the corresponding data structures and calls the check_delivery_cost function (discussed more in detail later) to know if it is time to go deliver
- Patrolling: Randomly selects a tile that is in the surrounding area and creates the subIntention to reach it, the area size depends on the dimensions of the map
- PlannedMove: Calls the planPath function to obtain a plan to reach the destination, defines the action needed by the executor and starts the plan.

Most important functions

- checkDeliveryCost:

```javascript
async function checkDeliveryCost() {
	myAgent.intention_queue.sort ... // sort by distance
	let ret = false
	if (PARCEL_DECADING_INTERVAL == 1000000) {
	return false // if PDI is set to infinite return false
	}
	let queue = Object.assign([], myAgent.intention_queue)
	if (queue.length == 0 && carriedQty()) {
	return true
// if the intention queue is empty and I am carrying something return true
	}
	let par = queue[0]
	if (par[0] != 'go_pick_up' || !carriedQty()) {
	return false
// if I am not carrying anything or the next intention is not go_pick_up
	}
	let speed = 1 / (MOVEMENT_DURATION / 1000)
	let xypar = { x: par[1], y: par[2] }
	let dmp = distance(me, xypar)
	await nearestDelivery(me)
	.then(async (nearest_me) => {
	await nearestDelivery(xypar)
	.then(async (nearest_xypar) => {
		let dmd = distance(me, nearest_me)
		let ddp = distance(nearest_me, xypar)
		let dpd = distance(xypar, nearest_xypar)
```

```
                let carried = carriedReward()


                let second;

                await reward(par, nearest_me)
                .then((r) => {
                        second = carried/(PARCEL_DECADING_INTERVAL/1000) -
carriedQty() * (dmd / speed) //deliver now
                })
                if (carried*(PARCEL_DECADING_INTERVAL/1000 - carriedQty() * ((dmp
+ dpd) / speed) < second) {
                        ret = true
                }
                else {
                        ret = false
                }
        })
        })
        return ret
}
```

- PlanPath:

JavaScript

```
// Create a plan to move from the starting position to the end, to do so it
first call setBaseKnowledge, it then creates a pddlProblem and asks
onlineSolver for a solution,
// the domain needed is defined in the file domain-deliveroo.pddl
async function planPath(start, end) {

        let plan;
        setBaseKnowledge(start)
        var pddlProblem = new PddlProblem(
        'deliveroo-problem',
        believes.objects.join(' '),
        believes.toPddlString(),
        'and (at me t' + end.x + '_' + end.y + ')'
        );
        let problem = await pddlProblem.toPddlString()
        let domain = await readFile('./domain-deliveroo.pddl')
        .then(async (dom) => {

        plan = await onlineSolver(dom, problem);
```

```
        })
        .catch(() => {
        console.log('server busy')
        })
        agents = new Map()
        return plan
}
```

- SetBaseKnowledge:

```
JavaScript
// Create a knowledge base needed to create a plan, the knowledge base
includes the map believes, the current position of the agent, and the tiles
currently occupied by other agents
function setBaseKnowledge({ x, y }) {
        believes = new Beliefset()
        for (const e of map_believes.entries) {
        believes.declare(e[0])
        }
        believes.declare('at me t' + Math.round(x) + '_' + Math.round(y))

        for (const a of agents.values()) {
        believes.declare('occupied t' + Math.floor(a['x']) + '_' +
Math.floor(a['y']))
        }

        return true
}
```

- **PDDL Domain:**

This is part of the domain file with the definition of the possible predicates and one of the possible actions (the others are almost identical other than the specific predicate used . The only noticeable decision taken in this part is the use of only right and down predicate with left and up defined swapping the order of the arguments where needed.

```
;; domain file: domain-lights.pddl
(define (domain default)
        (:requirements :strips)
        (:predicates
        (tile ?t)
```

```
        (occupied ?tile)
        (delivery ?t)
        (agent ?a)
        (parcel ?p)
        (me ?a)
        (at ?agentOrParcel ?tile)
        (right ?t1 ?t2)
        (down ?t1 ?t2)
        )

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

Double agent

The general structure of the double agent is very similar to the single one.
The only added data structure is a Map of predicates called ‘locked’ that is used to avoid overlapping intentions between the two agents that might cause interferences.

The strategy defined for the collaboration is pretty straightforward: there are no predetermined areas of interest for the two agents, when one of the two sees a parcel it will calculate the possible reward it can gain from it (this is an optimistic estimation since it considers picking up said parcel and going straight to deliver), and asks the other agent to do the same, the one with the highest reward locks the parcel, putting it in the locked Map, and adds it to it’s queue. This type of interaction interferes with the normal queue structure of the intention_queue since the agent that receives the information of a parcel somewhere in the map might be doing something else and moving and we do not know where in the queue the new parcel is being inserted, keeping this in mind when the intention loop progresses instead of just taking the first predicate in the queue the agents first sorts them by distance and then take the first one.

The main differences between the single and the double agents are:

- the presence of the onMsg events, these are used to exchange information between the double agents, such as their name and id, parcel positions, if an agent is currently looking for work, and eventual rewards to evaluate the assignment of the pickup intentions.
- the setting and checking of the locked Map to avoid two agents acting on the same intentions, the management of this is pretty simple as it’s just putting the intention to pickup a certain parcel in the Map when the agent is assigned it, we don’t really need to worry about removing the parcels since the key in the map is the parcel id which is supposed to be unique.

Results

Running the two scripts on the different challenges with 5 minutes each led to these results:

	challenge	1	2	3	4
single	2	290	25	1358	505
double	3	1104	2321	724	

As visible from the numbers the agents work fairly well with the exception of the second challenge for the single agents, this is due to the very fast parcel spawn rate and low mean reward set in the level,with the consequent inability of the agent to decide which parcel to pick up and ask the onlinesolver for a plan before the parcel disappear, this might be solved by using an offline solver reducing drastically the time needed to come up with the plan.