Excercise 3
Implementing a deliberative Agent

Group №: 272257, 262609

October 20, 2020

1 Model Description

1.1 Intermediate States

The states are described by the following attributes:

- logist.topology.Topology.City *city*, the city the agent is currently in,
- LinkedList<Task> *ctask* (default value = *empty*), that represent the list of tasks the agent has picked up. Note that the elements of the list also have 3 attributes : *pickupCity* that represents the city the agent can pick up the contract at, *deliveryCity* that represents the city the agent must deliver the contract at, and *weight* that represents the weight of the contract,
- LinkedList<Task> *free_tasks* (default value = *tasks present in the environement*), that represents the list of tasks that have not been picked up yet. The elements of the list have the same attributes as in the list mentioned before,
- double *cost* (default value = 0), the sum of all steps needed to get to this state,
- a method long *cweight*(), that returns the sum of the weight of the current tasks.

1.2 Goal State

A goal state is defines as a state at with all tasks have been picked up and delivered. Translated , this gives the condition $ctask.size() == 0$ && $free\_tasks.size() == 0$.

1.3 Actions

We distinguish three types of steps (that can be though of as an a edge between node-states in the graph). **Move-edges** that represent the physical movement of the agent (a change to a *neighbor* city) and a have a cost equal to the distance between the two cities, **Pickup-edges** that represent the picking up of a task (and therefore can only happen in a city where a task is available) and have no cost (but has an influence on the carried weight of the agent, its remaining-capacity, and available tasks) and **deliver-edges** that represent the act of delivering a task, those have no cost but have an influence of the carried weight of the agent and its remaining-capacity.

Note that this generate a lot of nodes at each iteration. In practice, in the A^* and BFS, we always choose the **deliver-edges** when it is possible. This reduces consequently the number of possible states at certain times. It seems also intuitive to deliver a task when said task has been picked up and the agent happens to pass through its delivery city. Not doing so would automatically be non-optimal. That is why this restriction actually helps us in the plan making process.

2 Implementation

We implemented those two algorithms in the *Deliberative* class. We suppose that we begin at the city *city*.

2.1 BFS and A*

Algorithm 1: BFS and A^*

```
initialization: Set $Q$ a queue of $State$ objects with one element $state$ at default values with
                $state.city = city$ and $state.heuristic = true$ if $A^*$ and $false$ if $BFS$ and set $C$ to
                be an empty queue.
$s \leftarrow nil$
while $Q.size() \geq 1$ and $s$ is not a goal state do
    $s \leftarrow Q.pop()$
    if $s.goal()$ then
        return $s$
    end
    if $s.rajoute(C)$ then
        $Q \leftarrow Q + s.succ(topology, capacity)$
        $Q \leftarrow Q + succ(s)$
        if Algorithm is $A^*$ then
            $Q.sortBy(f)$
        end
        if Algorithm isn't $A^*$ then
            $Q.sortBy((s' \mapsto s'.cost))$
        end
    end
end
return $nil$
```

where the functions $s.rajoute(C)$ takes a queue $f(s) = s.cost + s.heuristic()$.

2.2 Heuristic Function

The heuristic function is given by

$$s.heuristic() = \begin{cases} max\{s.city.distanceTo(task.pickupCity)+ \\ \quad +task.pickupCity.distanceTo(task.deliveryCity)\}, \\ max\{s.city.distanceTo(task.deliverCity)\}. \end{cases}$$

where the maximum is taken on the set

$$\{task \in s.free\_tasks\ \}$$

in the first case when it is not empty, and on

$$\{task \in s.ctask\}.$$

in the second case, when the first set is empty. We claim that such an heuristic is optimal for the A^* algorithm. Indeed, one easily sees that in order to get to a goal state, one needs at least to pick and deliver the remaining tasks that were not picked up before, especially the farthest one. This is the exactly the value *s.heuristic* takes in the first case. In the second one, assuming that all the tasks were picked up, one still needs to deliver each tasks that has yet to be deliverde. In particular the farthest one. This

corresponds to taking this maximum in the second case.
As the seen in the course this allows us to say that A^* does find the optimal plan. This heuristic function was inspired by trying to force the agent to take a maximum of tasks and the deliver them.

3 Results

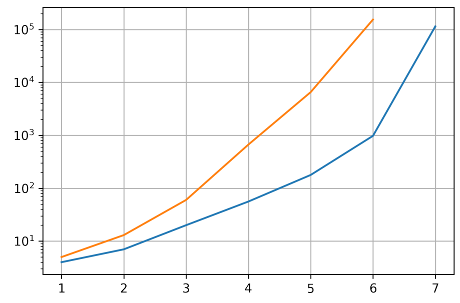

Figure 1: Compared running times of BFS (orange) and A^* (blue) (y axis in log scale, milliseconds).

3.1 Experiment 1: BFS and A* Comparison

3.1.1 Setting

We run simulations for BFS and A^* agents, with seed 2342356 on the "Switzerland" map and raise the task number until running time $T(n_{tasks})$ until $T(n_{tasks}) \geq 1min$.

3.1.2 Observations

We observe that BFS grows clearly faster than A^* that we are able to optimize for 5 tasks with BFS and 6 tasks with A^* in under a minute. We also observe that the running times grows faster than $O(e^n)$ for both algorithms (see Figure 1).

3.2 Experiment 2: Multi-agent Experiments

3.2.1 Setting

We run simulations (again with seed 2342356 on the "Switzerland" map) with a varying number (1 to 3) of A^* agents to observe their interactions when optimizing on the same state-space. In order to have some point of reference we run the same experiment with *naive* agents as well.

3.2.2 Observations

	NAIVE AGENT	SINGLE ASTAR	TWO ASTAR AGENTS	THREE ASTAR AGENTS	TWO NAIVE	THREE NAIVE
total distance	2590,0	1010,0	1610,0	2310,0	3860	5260
Distance Improvement factor	1,00	2,60	1,60	0,90	0,67	0,49
Time of last action	32077776389	14522221631	11499999499	10899999545	23499998977	21499999077
Time Improvement Factor	1	2,21	2,79	2,94	1,36	1,49

We observe that if we optimize for distance, a single A^* agent is more efficient than any other tested scenario (here we consider total distance traveled by all the agents in the scenario). Adding other A^* agents reduces performance. This cannot be said if we optimize over time, then adding more agents is actually beneficial (but we observe that the more agents we add the less improvement we see).