Excercise 3
Implementing a deliberative Agent

Group №: 272257, 262609

October 20, 2020

1 Model Description

The problem studied in this report is once again the pickup and delivery problem. This time, our goal is to implement a deliberative agent. This agent establishes a complete plan before doing anything. The plan is supposed to render the best efficiency in terms of cost to deliver the given tasks. Once the plan is established, the agent will act accordingly. If only one agent is present on the tracks, the agent will execute the plan in its entirety and will stop, as nothing has to be done anymore. If two or more agents are present, it can be that one agent disrupt the plan of the other. What happens in that case is that the agent which finds its plan unexecutable will compute one once again with updated informations about the environement when encountering an undoable action (ex: picking up a contract that has already been picked up by an other agent earlier). To establish the plan, we will use research algorithms *BFS* and *A** on trees. The nodes of the trees will be implemented by the class *State*.

1.1 Intermediate States

Here we integrated the intermediate states with the following attributes:

- logist.topology.Topology.City *city*, the city the agent is currently in,
- LinkedList<Task> *ctask*, that represent the list of tasks the agent has picked up. Note that the elements of the list also have 2 attributes : *pickupCity* that represents the city the agent can pickup the contract at, *deliveryCity* that represents the city the agent must deliver the contract at, *weight* that represents the weight of the contract,
- LinkedList<Task> *free_tasks*, 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 mentionned before,
- State *parent*, that is the predecessor of the current state. This will be used after the application of the *BFS* and *A** algorithms to build back the plan from the goal state we find using those algorithms,
- double *cost*, that is equal to the sum of all steps needed to get to the state,
- Act *act*, represents the action taken at the given state. This value can vary between *START*, *PICKUP*, *MOVE* and *DELIVER*. The meaning of these actions will be given in a section below.
- int *depth*, that represents the depth of the nodes in the tree,
- Boolean *heuristic*, that represents the method used by the search algorithm. If $heuristic = true$, then the *A** algorithm is used. Otherwise the *BFS* algorithm is used.

1.2 Goal State

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

1.3 Actions

The agent has 4 different possiblities of action. An action is implemented by the *act* variable and it defines what the chilren of a node can be. We provide a clearer explanation. Suppose we have a state *state*. We will only give the attributes that change, the rest is assumed to remain constant except for the *parent* attribute that is set to *state* and *depth* that is set to $state.depth + 1$ for the child. Note that all combination of mentioned cases can happen and thus we generate a child node for each individual possibility if not mentionned otherwise.

- If $state.act = START$: Then we can have

$$s'.act = \begin{cases} PICKUP & , \text{ if } state.city \in \{task.pickupCity \mid task \in state.free\_tasks\}, \\ MOVE & , \text{ in any case.} \end{cases}$$

- If $state.act = PICKUP$: Note that it means that we have that there exists $task \in state.free\_tasks$ such that $task.pickupCity = state.city$. Then we can have

$$\begin{aligned} s'.ctask =& s.ctask.add(task), \\ s'.free\_tasks =& s.free\_tasks.remove(task), \\ s'.act =& \begin{cases} PICKUP & , \text{ if } state.city \in \{task.pickupCity \mid task \in state.free\_tasks.remove(task)\}, \\ MOVE & , \text{ in any case,} \\ DELIVER & , \text{ if } state.city \in \{task.deliveryCity \mid task \in state.ctask\}. \end{cases} \end{aligned}$$

- If $state.act = DELIVER$: Note that this means that we have a task $task \in state.ctask$ such that $task.deliveryCity = state.city$. Then we can have

$$\begin{aligned} s'.ctask =& state.ctask.remove(task), \\ s'.act =& \begin{cases} PICKUP & , \text{ if } state.city \in \{task'.pickupCity \mid task' \in state.free\_tasks\}, \\ MOVE & , \text{ in any case,} \\ DELIVER & , \text{ if } state.city \in \{task.'pickupCity \mid task' \in state.ctask.remove(task)\}. \end{cases} \end{aligned}$$

- If $state.act = MOVE$: Then for each $city'$ such that $state.city.isNeighbour(city')$, we can have

$$\begin{aligned} s'.city =& city', \\ s'.cost =& state.cost + state.city.distanceTo(city'), \\ s'.act =& \begin{cases} PICKUP & , \text{ if } city' \in \{task.pickupCity \mid task \in state.free\_tasks\}, \\ MOVE & , \text{ in any case,} \\ DELIVER & , \text{if } city' \in \{task.deliveryCity \mid task \in state.ctask\}. \end{cases} \end{aligned}$$