Excercise 3
Implementing a deliberative Agent

Group №67 : Samuel THIRION, Jonas Schweizer

October 20, 2020

1 Model Description

1.1 Intermediate States

A state, implemented in the class `State`, is represented by the current city of the agent (`currentCity`), the list of tasks carried by the agent (`tasksInVehicule`), and the list of tasks still available to pickup on the map (`tasksRemaining`). Each state has a `clone()` function that allows to create the neighbor states. At all time, each state has the updated variable `costToReach` holding the cost of the path (= list of transition) that has lead to it.

1.2 Goal State

A final goal state is a situation were all the packages have been delivered, therefore both `tasksRemaining` and `tasksInVehicule` need to be empty. We call the function `allPacketsAreDelivered()` on the state which tells us whether the state is a goal state or not.

1.3 Actions

In a state, there are several possible `Transitions`. They have several variables: `action` which is an enumerate with values `PICKUP`, `DELIVER` or `MOVE`; `task` which is the involved task (only if `PICKUP` or `DELIVER`), `destination` which is the delivery city of the task or the destination of the move, and `cost`, which is 0 if `PICKUP` or `DELIVER` and the cost of the move ($distance * costPerKm$) if `MOVE`.

The function `getTransitions` returns the list of possible actions, which are:

- if the vehicle has a package to his current city, `DELIVER` is the only action. It is obvious to see that any other action would be a waste of time, since delivering a task comes at no cost.
- else, it can either `PICKUP` an available task in the current city or `MOVE` to a neightboring city.

To compute the `nextState`, which results from taking the transition `t` in `state`, we use `y.getNextState(state)` which clone `state`, applies the transition `t` and returns the result.

To represent a path, we use the class `TransitionList` which is a list of `Transition`.

2 Implementation

2.1 BFS

We impemented a BFS as pictured in the lecture. One difference lies, though : as the graph as weights, it is possible to visit a state with a certain cost once, and to try to visit it with an inferior cost a second time. If we stick to the classical version of BFS, the second visit is ignored and we may miss the optimal

solution. To avoid that, the cost of a visited state is remembered. When the state tries to be visited again, we compare the new cost to the cost of the previous visit, and visit the state again if it is reached by an inferior cost than that of its copy in `C`. Once a goal state is reached, the list of transitions leading to it is added to the linkedlist `allGoalPaths`. Once the graph is entierely visited, we pick up the best path (path with smaller cost) in `allGoalPaths`.

2.2 A*

The plan using A* is computed using the function `runASTAR()`. It is encapsulated in `processASTARPlan()` which measures the computing time and the memory used by the runtime once the A* algorithm completes. As `runASTAR` returns the best path (list of transitions), it also computes the plan from it. `runASTAR()` follows exactly the algorithm seen in the lecture. The priority queue stores the states, and sort them using the state comparator `State.SortByEstimatedCost()` (itself using one of the heuristic functions described bellow).

2.3 Heuristic Function

We want to estimate the cost `h(state)`, which is the minimal cost to reach a goal state starting from `state`. One can prove that A* always finds the optimal solution as long as `h()` never over-estimates the true minimal cost `h*(state)`.

2.3.1 Minimum Spanning Trees (MST)

In a given state and in order to reach a goal state, it is certain that the agent has to go in every city in which a task (picked up or not) still has to be delivered and in every city which has a task waiti ng to be picked up. Knowing that, lets define the subset of cities $A = \{currentCity\} \cup \{all\ cities\ yet\ to\ be\ visited\ at\ least\ once\}$. We define the complete graph G with the vertices A, and the weighted edges as follow : $w(e) = distanceBetweenCities(e.cityA, e.cityB)$ which is the minimal distance between the two cities of the edge. Thanks to Kruskal's algorithm, we compute the Minimum Spanning Tree[1] of this graph and claim that its total cost is inferior to the true cost `h*(state)`.

Indeed : let's be $P = (e_1, e_2, ..., e_n)$ a path in G leading to a goal state and whose cost is minimal. We can then remove edges from P (edges involved in cycles, edges appearing multiple times, if any) until P becomes a tree of G, named T. As the weights are non negative, we thus have $cost(MST) \leq cost(T) \leq cost(P) = h^*(state)$. It is thus proven that this heuristics never over-estimate the true cost, and that A* always returns the optimal plan.

Summary : `h(state)` = cost of MST of the complete graph whose vertices are A. Even if the MST can be computed in $O(|A|^2 log(|A|))$, our observations show that the computation time of `h(state)` is high and slows the A* implementation down. We use a much simpler heuristic instead, introduced in the following section.

2.3.2 Maximal Distance

For every task t still on the map, the minimal distance to travel is $distance(currentCity, t.pickupCity) + distance(t.pickupCity, t.deliveryCity)$. For every picked up task t, the mimimal distance to travel is $distance(currentCity, t.deliveryCity)$. We thus define `h(state)` = maximum of those distances, always inferior to the true cost `h*(state)`. This heuristic is used in our implementation of A*, as it is faster to compute than a MST.

[1] A tree connecting all vertices of G whose cost is minimal

3 Results

3.1 Experiment 1: BFS and A* Comparison

The limitation of BFS is the memory and the processing time. Indeed, such algorithm requires to keep in memory a lot of states, and explore all possible states to find out the optimal goal state and how to reach it. Astar is an optimization which helps us to find faster and with less memory the optimal goal.

The next figure shows the log of the processing time according to the number of tasks. We see that it evolves exponentially, so the time explodes fast for BFS (at 10 tasks) and a bit after for ASTAR (between 11 and 12 tasks).

For more tasks than 12 tasks, it doesn't compute in less than 1 minute. What is interesting is that the memory needed is as expected, big for BFS (more than 1.2Gb for 9 tasks) and a little bit smaller for ASTAR (700Mb for 9 tasks). It seems to follow the same evolution as the computation time.

To conclude, both techniques requires a lot of computational power, but ASTAR in addition to finding the optimal solution as BFS, requires a lot less time and memory to compute (it is approximately 4 times faster than BFS).

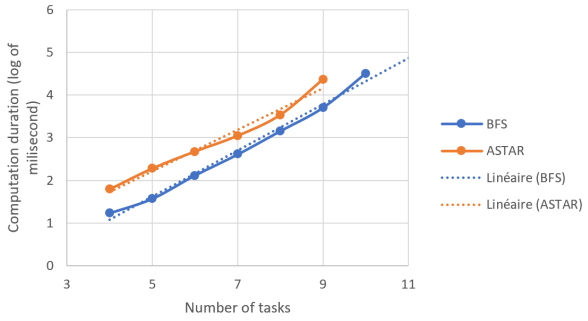

Figure 1: Computation time in log scale against number of tasks

3.2 Experiment 2: Multi-agent Experiments

3.2.1 Setting

The settings are simple: 9 tasks need to be delivered in Switzerland (`seed`=23456, with probability distribution uniform between 0 and 1, reward constant with short-distance 100 to 99999 and weight constant to 3).
We will run 1, 2 and 3 agents and see how well they perform together. For each simulation, we wait until all tasks are delivered are measure the *efficiency* as the stationary value of the `reward per km`. Indeed, as the cost per km and the total reward of all the task are constant accross the simulations (for the same seed), the reward per km achieved once all tasks are delivered is a function of the total cost of solution. The higher the cost, the less the reward per km.

3.2.2 Observations

As the number of agents increases, the global efficiency increases and the efficiency per agent decreases. This is natural as each agent will deliver a subset of all the available tasks : the overall work is splitted. However, the agents do take into account the presence of other agents when computing their optimal plan. The immediate consequence is that the individual efficiency drops a lot as the number of agents increases. With other-agent awareness, we could hope to have a constant individual efficiency and a much better global efficiency.

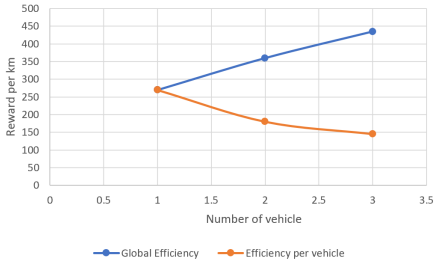

Figure 2: Efficiency for 1, 2 and 3 agents