Exercise 2: A Reactive Agent for the Pickup and Delivery Problem

Group №17: Thibauld Braet, Nathan Greffe

October 9, 2018

1 Problem Representation

1.1 Representation Description

State representation: the pair *(current city; city the task is headed to (could be null))*. An agent can only see the city it is currently in and one of the tasks that are present. In each state you have the possibility to:

- pick up the package (if there is one) and let the system deliver it along the shortest path.
- reject the package and move to a neighbouring city (which is decided by Value table)

So for a topology with n cities, when the decision has been made to move to another city, we can end up in *1+(n-1)* possible states. The immediate rewards are computed by $delivery\_reward - distance * cost\_per\_km$ or simply $-distance * cost\_per\_km$ if we don't carry a package. The probability transitions from state *(city A, city B)* to state *(city B, city C$\|$null)* is simply given by the probability to have a package to *city C$\|$null* in *city B* (obtained from the Logist API).

1.2 Implementation Details

Our Q-table is 2 dimensional (#states rows and #actions columns). Each vehicle has his own Q-table because the rewards and costs for each vehicle might be different depending on the cost per kilometer or the capacity of the vehicle. The reactive agent has only one vehicle (moodle forums) but the API does not specify there could not be several so the code handles that case.

The states are defined as integers, we wrote functions to go from cities ids to state integer and vice-versa. We do not implement states of the form *(city A, city B)* if there are no tasks from *city A* to *city B*. This was initially thought of to optimize the costs, it is not really useful in Logist since the probabilities are (statically) always different from 0 but could be in a more general setting. Regarding the actions, the action "going to a non-neighbouring city" exists in ram (to ease action indexing) but we do not iterate over it, neither for the QTable update nor for the optimal action selection. We keep iterating until the biggest absolute value of the difference between a state and his update is smaller than 1×10^{-7} (very small value but that does not need too many iterations either). We also check if the vehicle has the capacity to take the task. The rest of the implementation is simply the application of the value-iteration algorithm.

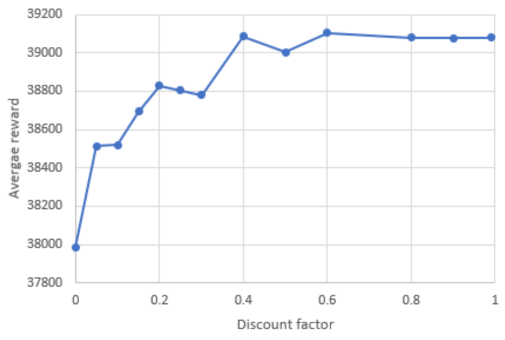

Figure 1: Average reward in function of discount factor

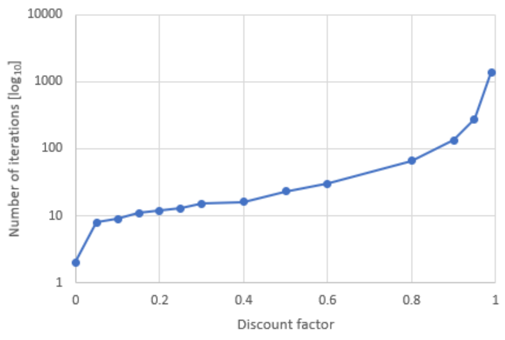

Figure 2: Number of iterations in function of discount factor

2 Results

2.1 Experiment 1: Discount factor

2.1.1 Setting

We used several agents, all traveling in the given topology of France. The tasks had a uniform probability distribution where the short-distances policy was applied to determine the reward. Each task was given a weight of 3 and since each agent had a capacity of 30, this was not an important factor. Each agent also travelled at the same speed and with the same cost per km. The hometown differed but the algorithm for convergence of the State Values does not attach importance to the beginning state.

2.1.2 Observations

It can be seen that the agent benefits from taking into account possible future rewards. However, this value flattens and even descends a bit when going to 1. Although it is not represented in this graph, the optimum value for the discount factor probably lies around 0.5. This is probably due to noise. These results (see Figure 1) are quite surprising, indeed, since the problem is known, the solution should be optimal with respect to the chosen value of γ and the closer γ is to 1, the closer the objective is to maximizing the average reward. We found out why by exploring our QTables, by looking at them, we understood that, even with high γ, the policy is to nearly always take the package. Thus, all values of γ implements nearly the same policy and have nearly the same results (due to noise and the few cases in which the optimal choice is not to be greedy). It would have been better to observe that the average reward increases with γ but we couldn't find proper settings in Logist to have such a problem (we tried several topologies and a lot of combinations of parameters but with no success).

We also noticed that the larger the discount value, the longer (exponentially) it takes the algorithm to converge (see Figure 2). This makes sense since the reward t steps further from now is reduced by a factor $(\gamma)^t$. When using a discount factor of 1, the algorithm even never converges as stated in the course.

2.2 Experiment 2: Comparisons with dummy agents

2.2.1 Setting

We compared three agents with each other. *Reactive* is the agent as in experiment 1 using 0.95 as discount factor. *Random* is the agent that was given as template. *Dummy* is a new agent that behaves as follows. When no task is available, it moves to the neighbour where the expected reward (i.e. the weighted sum of the reward of the packages, without taking the distances to travel or the cities he will en up in into

account) is the highest. If there is a task available it looks if a neighbour has a higher expected reward. If so, then it moves to that neighbour. Otherwise it picks up the task. So we don't take into account distances.
The same settings were used as in experiment 1 such that no difference can be made in capacity or cost per km.

2.2.2 Observations

There can be noticed that the reactive agent still performs better than the other ones, as expected. When looking at the QTable of the reactive agent, we notice that it is almost always most beneficial to pickup the task. The other agents will drop a task much more quickly and thus perform worse. E.g. when the pPickup value of the Random agent is set higher, it performs way better because it will almost always pickup the available task.

Agent	Reactive	Random	Dummy
Average reward	39228	18748	37204

Table 1: Average reward of different agents

2.3 Experiment 3

2.3.1 Setting

While implementing the algorithm. We thought about, rather than computing the states values on the fly, storing them at the beginning of every iteration and then updating the values of the states all at once. The difference between these two approaches is that, at iteration k, if there is a path between $State_1$ and $State_2$, in the basic setting, $State_1^k$ would be used to compute $State_2^k$, whereas in the other setting, $State_1^{k-1}$ would be used to compute $State_2^k$. The first approach intuitively seems to convege faster but is not fair between every states (although that might not be a problem). These were tested on the map of the Netherlands (see *reactive2.xml*) instead of the map of France by accident but it does not invalidates the results in any way. The correctness was verified by comparing the QTables.

2.3.2 Observations

We ran our experiments with 3 discount factors (0.99, 0.95 and 0.9). Our results were that the basic method (*basicValueIteration*) took (1155, 229, 113) and the second one (*alternativeValueIteration*) (1958, 384, 187) iterations to converge, the convergence rate of the first method is thus the best in term of number of iterations. However, in the second implementation, we computed $\gamma * \sum_{s' \in S} T(s, a, s') * V(s')$ once for every city, which allowed us to improve speed, indeed the computations times were (1324, 548, 311) and (432, 199, 127)ms. The second method is thus faster. It is possible to compute $\gamma * \sum_{s' \in S} T(s, a, s') * V(s')$ less than once per non-zero entry in the QTable for the first approach as well[1]. Once we used that optimization (*basicValueIterationV2*), the times were reduced to (359, 153, 104)ms, which is the smallest of the 3 cases (as expected, the number of iterations did not change).

[1]It is possible to compute it twice per city since the s' are ordered by pickup city and we cannot go from a city to itself. We can thus update the values once in the beginning and once when all of the $V(s')$ corresponding to a city were updated (i.e., once our current state is related to a new pickup city)