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

Group №23: Jean-Thomas FURRER, Emily HENTGEN

October 10, 2017

1 Problem Representation

1.1 Representation Description

In our implementation, a state is described by two different attributes: the current city of the agent and the destination city of the task. Implicitly, whether there is an available task in this city is indicated by the destination city, and a *null* value means there is no available task.

An action is described by three different attributes: whether it consists of a pick up or just a move between two cities, the departure city and the destination city.

The reward table consists of a table mapping a *State* to a numerical reward. Similarly, the probability transition table maps a *State* to the probability of arriving at this *State*. For a *State* with an available task, this is simply the probability of having an available task in the current city to the destination city, given by the task distribution. For a *State* with no available task, that is, a state with a *null* destination city, this is the probability of having no task in this city, also given by the task distribution.

1.2 Implementation Details

The agent behaviour is implemented in the *ReactiveTemplate* class. For comparison purposes, there are two additional dummy agents: a random agent who randomly travels between cities, and randomly picks the available task or not (implemented in the *RandomTemplate* class) and a routine agent, who also randomly picks up tasks, and otherwise follows a randomly predetermined itinerary (implemented in the *DummyTemplate* class). The two classes *State* and *Action* (*template.Action*) implement a state respectively an action representation, both defined by their respective above-mentioned attributes.

The main data structures are implemented as *HashMap*s, so that accessing elements is fast. The data structure representing the set of all states is a *List*. The additional data structure *statesForCity* is not strictly necessary, but speeds up the retrieval of all states associated to a given city (instead of looping through the entire *allStates* list).

rewards *HashMap<template.Action, Double>*

probabilities *HashMap<State, Double>*

bestActions *HashMap<State, Action>*

bestValues *HashMap<State, Double>*

allStates *List<State>*

statesForCity *HashMap<City, List<State>>*

The redundant attributes between a *State* and an *Action* (in particular the current city/departure city) make the use of the HashMaps more practical: the keys, which conceptually consists of a *(State, Action)* pair are here either a *State* or an *Action*, which simplifies the overall implementation.

Algorithm 1 Learning Strategy (Value iteration)

INPUT
rewards ▷ the table mapping an action to its expected reward
probabilities ▷ the table mapping a state to the probability of being in this state
allStates ▷ the set of all possible states
statesForCity ▷ the table mapping a city to its set of possible states
discountFactor ▷ the probability an agent picks up a task
OUTPUT
bestMoves ▷ the table mapping a state to the next best action and the associated value

$hasConverged \leftarrow false$
initialize $bestMoves$ to an empty table

while not $hasConverged$ **do**
  for $state$ in $allStates$ **do**
    $maxQ \leftarrow -\infty$
    for $action$ in $state.actions$ **do**
      $acc \leftarrow 0$
      for $nextState$ in $statesForCity[action.cityTo]$ **do**
        $acc = acc + probabilities[nextState] * bestMoves[nextState].value$
      end for
      $Q = rewards[action] + discountFactor * acc$
      if $Q > maxQ$ **then**
        $maxQ \leftarrow Q$
        $bestMoves[state] = (action, Q)$
        $hasConverged \leftarrow false$
      end if
    end for
  end for
end while
return bestMoves

2 Results

2.1 Experiment 1: Discount factor

2.1.1 Setting

In this setting, three different reactive agents with a discount factor of 0.99, 0.5 and 0.0 respectively are tested against each other.

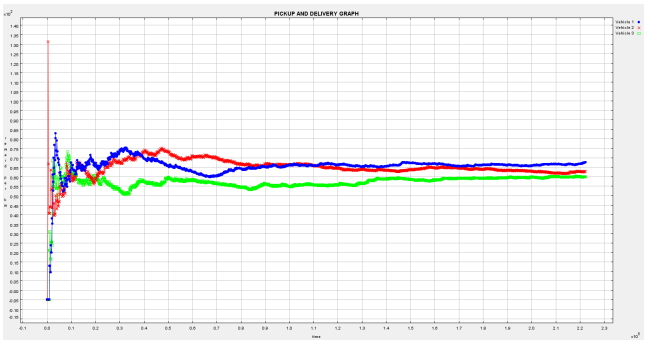
2.1.2 Observations

On average, the higher the discount factor, the higher the performance of the reactive agent: the reactive agent with a discount factor of 0.99 has in the long run a reward per km greater than all the other agents. The agent which displays the lowest performance is the one with discount factor 0.0, that is, the one which optimizes its behaviour for an immediate reward. These results however can often be observed only after a significant number of steps, due to important fluctuations between the average reward per km of the reactive agents. At some points during the simulation for instance, the reactive agent with discount factor 0.5 may outperform the reactive agent with discount factor 0.99, before the tendency eventually stabilizes (Figure 1).

2.2 Experiment 2: Comparisons with dummy agents

2.2.1 Setting

Here, a reactive agent with discount factor of 0.85 is first tested against a random agent and a routine agent who both accept tasks with probability 0.85. Two reactive agents with discount factors 0.99 and 0.5 are then tested

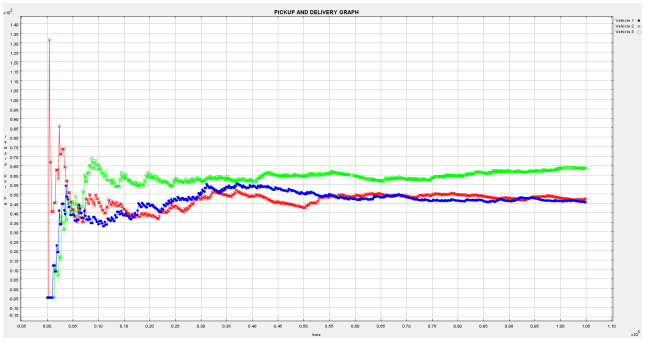


(a) Reactive agent 1 with discount factor = 0.99 (blue) – reactive agent 2 with discount factor = 0.50 (red) – reactive agent 3 with discount factor = 0.0

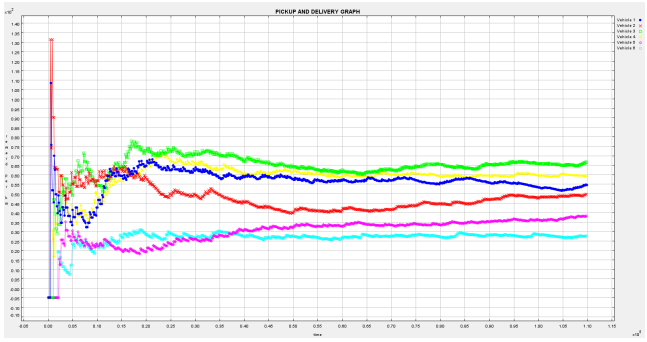
Figure 1: Discount factor influence on the reward per km

against two random agents and two routine agents accepting tasks with probability 0.99 or 0.5.

2.2.2 Observations



(a) Reactive agent with discount factor = 0.85 (green) against random dummy agent with discount factor = 0.85 (blue) and routine dummy agent with discount factor = 0.85 (red)



(b) Reactive agent with discount factor = 0.99 (green) against another reactive agent with discount factor = 0.5 (yellow), a random agent with discount factor = 1.0 (blue), a dummy agent with discount factor = 1.0 (red), a dummy agent with discount factor = 0.5 (magenta) and a random agent with discount factor = 0.5 (cyan)

Figure 2: Reactive agents performance compared to dummy agents performance

The reactive agent displays a higher performance than both the routine dummy agent and the random dummy agent when the discount factor respectively the probability they accept an available task is 0.85 for all three agents. Although the reward per km of both dummy agents fluctuates over time, their performance is rather similar in this case (Figure 2a).

When both the random dummy agent and the routine dummy agent systematically accept an available task, the reactive agent with discount factor 0.99 as well as the one with discount factor 0.5 still perform better. It is the random agent which accepts tasks with probability 0.5 which displays the lowest performance, but the random agent which systematically accepts task turns out to have a reward per km rather close to the one of the reactive agent with discount factor 0.5 (Figure 2b).

In the long run, a high discount factor tends to increase the reward per km of a reactive agent, since this agent seeks an optimal strategy that gives an important weight to the future. On the other hand, the performance of a reactive agent with a low discount factor, which optimizes for immediate rewards, comes close to the one of a random agent which systematically accepts tasks. The learning performed beforehand by a reactive agent still gives him a slight advantage over a random agent though.