Ecole Polytechnique Fédérale de Lausanne - EPFL

CS-430 Intelligent Agents

Mission 2

Reactive Agent

Armand Bosquillon de Jenlis
Vladimir Somers

Group 27

Professor : Boi Faltings

Academic Year 2015 - 2016

1 Design Choices

In this section, we are going to describe the state representation of the given problem and the structures we chose to build the different tables used in the Value Iteration algorithm. We will also explain the way these structures are implemented in Java. When we implemented the problem, we tried to always optimize memory consumption and time complexity. This is why we used arrays, ArrayLists and HashMaps instead of simple arrays. Finally, we will use very often the same notation for principal variables: **n** is the number of cities, **s** is any state and **a** is any action.

1.1 State representation: s

In order to choose the state representation, we have to think about all the information an agent is given when arriving in a new state (when the "act" method of the reactive agent is called). The two most important informations are the current city and the deliver city of the available task. Two other informations are the weight of the task and the capacity of the vehicle, but they do not change the state representation(we will talk about them later, see Section 1.9). Since we have n cities and n possible cities for thedestination of the available task, that gives us n^2 possible states.

We decided to represent the states by integers in the interval $[0\ ,\ n^2 - 1]$. If s is the value of a state where $s \in [0, n^2 - 1]$, then:
CityFrom $= s\%n$, is the index of the city where the agent is located.
CityTo $= s/n$, is the index of the destination city of the available task. If CityTo = CityFrom, then it means that no task is available in this state. You can see the schematic state representation on Figure 1.

City To \ City From	**0**	**1**	**...**	**n-1**
0	s = 0	s = 1	...	s = n-1
1	s = n	s = n+1	...	s = 2n-1
...	...	...	...	...
n-1	s = (n-1).n	s = (n-1).n+1	...	s = n.n-1

Figure 1: State representation

1.1.1 Implementation

We represent the states by integers. We implemented two functions taking as argument the state number and returning the CityFrom index and the CityTo index.

1.2 Action representation: a

We represent an action a by an integer in the interval $a \in [0, n]$. Given the current state s, the **possible** actions are :

- Move to the **neighbor** city with index $i \in [0, n-1]$. In this case, $a = i$.
- *Pick Up* the available task and go to CityTo $= s/n$. In this case, $a = n$.

If we are in a state s where CityFrom = CityTo, then there's no task available and the *Pick Up* action $a = n$ is not possible in state s.

1.2.1 Implementation

We represent the actions by integers. In order to iterate on the possible actions from a given state s, we built an array S for which $S[s]$ gives an ArrayList of possible actions in state s.

1.3 Reward table representation: $R(s, a)$

Our Reward table have entries only for possible pairs of state and action. The reward is the expected reward of the delivered task if there is one, minus the cost of traveling from CityFrom to CityTo. This cost is the number of kilometers traveled times the costPerKilometer value of the corresponding vehicle. The expected reward of a task is given by the table r(i,j), which is provided.

The final Reward table is therefore :

- $R(s, a) = r(CityFrom, CityTo) - distance(CityFrom, CityTo).CostPerKm$ if $a = n$
- $R(s, a) = -distance(CityFrom, a).CostPerKm$ if $a \in [0, n-1]$

1.3.1 Implementation

We represent the Reward table by a table R where the entry $R[s]$ corresponding to state s is a HashMap<Integer,Double>. Each key of this HashMap is a possible action of this state. Each associated value is the corresponding reward.

1.4 Transition table representation: $T(s, a, s')$

An entry T(s,a,s') of the table T gives the probability to arrive in state s' given that the vehicle is in state s and take action a. Given that the vehicle is in city *CityFrom* and take action a, the destination city *CityDest* is equal to a if $a < n$ and *CityDest* is equal to *CityTo* if $a = n$ (because in this case a is the Pick Up action). Thus T(s,a,s') is non null only if the *CityFrom'* corresponding to s' is equal to *CityDest*. So, given CityFrom' $= s'\%n$ and CityTo' $= s'/n$, the value of these non null entries is $T(s, a, s') = p(CityFrom, CityTo)$. $P(i, j)$ is a given table where $p(i, j)$ is the probability that in city i there is a task to be transported to city j.

1.4.1 Implementation

This transition table is in practice filled with a lot of zeros, so we decided to implement it with an array of HashMaps containing also HashMaps, in order to store only non null elements.

1.5 Accumulated value of a state: $V(s)$

The accumulated value of a state is stored in the table V[], we have V[s] = V(s). The initial value of this state does not affect the final results, because it always converge to the same values. A better initial value of this array only allows the algorithm to converge after fewer iterations.

1.6 Optimal policy $\pi(s)$

The optimal policy is stored in the table Best[] where $Best[s] = \pi(s)$. Best[s] gives the index of the best action to take in state s.

1.7 Discount factor γ

We use $\gamma = 0.9999$ because:
A Value Iteration algorithm with a discount factor that is close to 1 gives a solution which is close to the optimal long term solution.
In practice, the value we used leads to a running time which is not too long.
The policy and the Accumulated values found with our stopping criterion remains the same around $\gamma = 0.9999$, so it is highly probable that this policy is the optimal one or gives a total reward close to the optimal one.

However, a Value Iteration algorithm with a discount factor close to 1 takes more steps to converge to a solution.

1.8 Stopping criterion of the Value Iteration algorithm

We stop the Value Iteration when no value of the Accumulated Value Array $V(s)$ has been modified by more than **0.001** during a complete iteration. We made this choice because:
It leads to a running time which is not too long.
When we continue to make some iterations after this stopping criterion, the value of the states do not change a lot and the policy found remains the same thus, with this stopping criterion, it is highly probable that the policy we found is the optimal one or gives a total reward close to the optimal one.

1.9 Vehicle capacity and task weight

We also handle the case where a task weight is higher than the vehicle capacity. If a vehicle arrives in a city where the weight of the available task is higher than the vehicle capacity, then the vehicle considers that it is in a state with no task available.
In order to adapt the best policy to such cases, we decided to keep the same state representation and to consider (in the probability transition table computation) that the Pick Up action is not possible in a state s if the expected weight of the task is higher than the vehicle capacity.

2 Simulation Results

2.1 Comparison between the random reactive agent and our reactive agent

As we can see on Figure 2, by learning a strategy via the value iteration, our agent gets better results than the basic agent that you gave us.

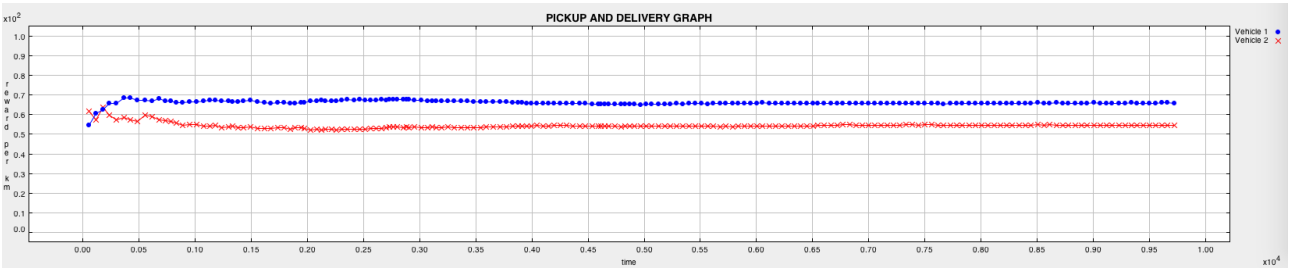

Figure 2: Evolution of the reward per km for our agent (blue) and the basic agent (red)

2.2 Results for two and three agents

The results are presented in figures 3 and 4. We can see the rewards per kilometer of each agent converge to the same value.

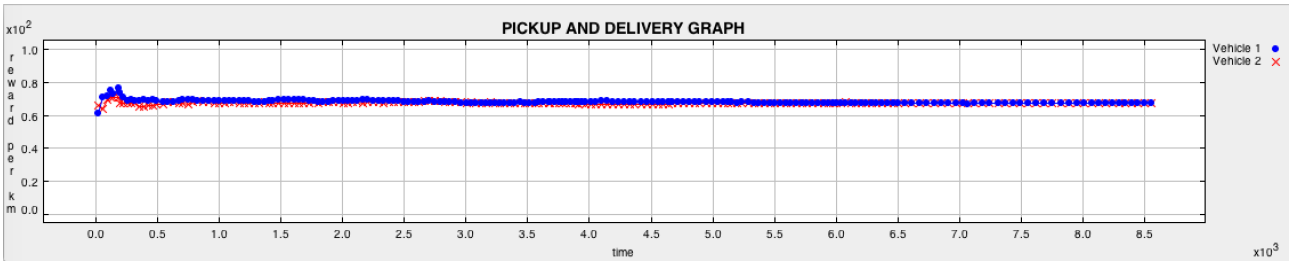

Figure 3: Evolution of the reward per km for two agents

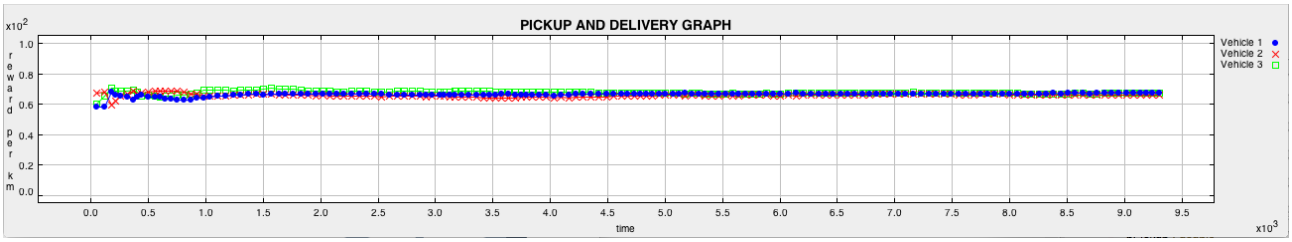

Figure 4: Evolution of the reward per km for three agents

2.3 Influence of the discount factor γ on the performances

The tabular below gives the asymptotic value of the reward per km R for different discount factor. As we can see in the tabular when the discount factor augments, R augments. When the discount factor is close to 0.9999, R does not evolve.

γ	reward per km
0.5	≈64.8
0.8	≈65.2
0.9	≈66
0.99	≈67.25
0.9999	≈67.25
0.99999	≈67.25