IA-reactive

Valentin Rutz, Ana Manasovska

October 2015

1 Representation of the world

- States S: Our state is represented by two parameters: the city we are currently in, and the available task that we potentially have there.If we have a task, the second parameter is its destination city or null if we don't have a task available. If C is the set of cities in the topology, the formal representation of the states is the following:

  $$S = \{i, j\} \text{ where } i \in C,\ j \in \{k | k \in C \text{ or } k == null\}$$

  Thus, the total number of states is $n(n+1)$, where $n = |C|$

- Actions A: Then, we have $n = |C|$ different actions:

  $$A = \text{moveto}(k) \text{ where } k \in C$$

  The action of picking up a package is simplified into moveto(k) when $k \in C$ is equal to the destination city of the generated package in current state $s = \{i, k\}$.

- Rewards $R(s, a)$:

  When a vehicle transports a package from city i to city j, it expects a reward for the package. This reward function was given by the table $r(i, j)$ in the TaskDistribution class.

  But for moving to a city j from i whether the vehicle is loaded with a package or not, it is expected that you pay a cost $c(i, j)$. This cost $c(i, j)$ should depend on the number of kilometers traveled by the vehicle and on a predefined cost per kilometer that depends on the vehicle itself since vehicle can be different, they consume more or less amount of resources to go from one city to another.

  A good profit function $R(s, a)$ would then be simply defined as follows:

  $$R(s,a) = \begin{cases} r(i,j) - c(i,j) & : k = j \text{ (pick up action)} \\ -c(i,k) & : k \neq j \text{ and } k \in N(i) \text{ (move to action)} \end{cases}$$

  where $a = \text{moveto}(k)$, $s\{i, j\}$ and $N(i)$ is the set of neighbors of i in C. The actions that cannot be taken from some state (do not satisfy any of the above two conditions) have the lowest possible reward ($Integer.MIN_VAlUE$) in order to not be possible to be taken.

- Transitions $T(s, a, s')$:

  $$T(S(i,j), a(k), S(m,l)) = \begin{cases} p(m,l) & : (j = k \text{ and } k = m) \text{ or } (k = m \text{ and } m \in N(i)) \\ 0 & : o/w \end{cases}$$

  We have defined the probability T(S,a,S') to be equal to the probability of S' (probability of generating a package from the start city in S' to the end city in S'). This value for T holds only in the cases when the transition from S to S' given action a is possible. In all other cases the probability is zero.

2 Description of the Code

2.1 Concrete Model

All of the abstract structures like tables $V(s)$, $R(s,a)$, $T(s,a,s')$ can be represented as Map from some key to some value. Here is a better look on how we represented all the abstract structures. The concrete type is HashMap<K, V> so thank you for having implemented the method hashCode in the class City.

- State: A state simply encapsulates the current location and the potential generated package (if no package is generated, the reference is null) and we create the state as State(City currLoc, City genPackage).
- Actions: Actions represents an action in our model which are "move to city k" $\forall k \in C$. It is represented by the class Actions(City city).
- RKey: RKey is a tuple of State and Actions that represents the key to the map $R(s,a)$. It contains a State and an Actions so its Java type is RKey(State s, Actions a).
- TKey: TKey is a tuple of State and Actions that represents the key to the map $T(s,a,s')$. It contains two States (the start and the end states) and an Actions so its Java type is TKey(State start, Actions a, State end).
- S: S is the set of all possible states so the Java type of S is Set<State>.
- $V(s)$: $V(s)$ is the easiest since the key is simply a state but the value needs to be both the best accumulated value (V(s)) and the action to take (best(s)) so that we can use $V(s)$ to learn $V(s)$ and we can use $V(s)$ to give the best action to the vehicle when it needs to act. Hence we represent the value of the Map by VValue(double value, Actions a) and its key by a State.
- $R(s,a)$: $R(s,a)$ is a map like $V(s)$ with key RKey(State s, Actions a) and value Double since it gives the reward of doing action a from state s.
- $T(s,a,s')$: $T(s,a,s')$ is a map like $V(s)$ with key TKey(State start, Actions a, State end) and value Double since it gives the probability of ending up in state s' doing action a from state s.

2.2 Setup

The setup method is called once per company since it has an Agent in its signature and allows us tho do the offline learning of $V(s)$. We initialize all sets and tables (S, A, $V(s)$, $R(s,a)$ and $T(s,a,s')$). For S, we simply loop over the cities in the topology, same for A. When those two are initialized, we can begin the real work.
We loop through all the states in S and actions in A to fill $R(s,a)$ according to the equation in part 1. We loop through all the states in S twice and all the actions in A to fill $T(s,a,s')$ according to the equation in part 1.
Filling $V(s)$ proves to be the hardest part since we need to use the given algorithm. The "good enough" condition is translated into the fact that if in iteration i the table is $V_i(s)$, then we stop when for all states $s \in S$, $|V_i(s) - V_{i+1}(s)| < \epsilon$ for some $\epsilon > 0$. This means that the difference between any two entries of the table only differs by an insignificant amount so we can stop trying to learn it. We use discount as γ for the weight of the future and then the pseudocode is very close to the actual implementation so there is nothing to add here.

```
Data: Vehicle v, Task t
Result: action for the agent
if t == null then
    s = State(c.currentCity, null);
    a = V.get(s);
    action = Move(a.city);
else
    s = State(c.currentCity, t.deliveryCity);
    a = V.get(s);
    if a.city = t.deliveryCity then
        /* Means we pick up the package                    */
        action = PickUp(t);
    else
        /* Means we refuse the package                     */
        action = Move(a.city);
    end
end
return action;
```

Algorithm 1: Vehicle's behaviour after completing an action

2.3 Act

The method act takes a Vehicle and a Task as arguments. When arriving in the method, we branch on whether the given Task is null or not (meaning there was no generated package).

If it is null, then we create a new State(currentCity, null) to describe the current state, we get the best action using $V(s)$ and we just move to the city given by the action retrieved from $V(s)$.

Else, we have a concrete generated package to a city $j \in C$. We describe our State(currentCity, generatedPackage) and then we retrieve the best action from V(s) which would be "move to city k".

If $k = j$, then we pick up the generated package (which automagically makes us move to k and deliver the package) and else, we refuse the package and we move to the city k anyway.