Reactive Agent: Model Solution

By Adam Richardson

Steps

1. Construct State and Action Representations
2. Build Data Structures and Methods
3. Code Q-Learning Algorithm
4. Logist

Steps

1. Construct State and Action Representations
2. Build Data Structures and Methods
3. Code Q-Learning Algorithm
4. Logist

States and Actions

- States and Actions defined in the context of a *Markov Decision Process*
  - Markov Process is a sequence of *Events*: what is an Event?
    - Definition of Markov Process: the probability of an event depends only on the previous event
      - Equivalent Definition: an Event is *the complete set of information required to know the probability of the next Event*.
  - Markov Decision Process: *Event* is decomposed into *State-Action pairs*.
    - Actions are deterministic given a State
    - How to decompose an *Event* into a *State-Action Pair* is problem-specific

States and Actions

- Actions: the degrees of freedom of an agent
  - The vehicle can move to a neighboring city
  - The vehicle can pick up the task in the current city and deliver it at the destination city.
- States: the information necessary to choose the action in order to make the State-Action Pair a complete Event
  - Current City: tells us what are the possible cities to move to
  - Task Available in current city: tells us whether or not we want to pick up the task
    - Could be no task

States: Coding

```
private class myState {
    public City currentCity; The Current City component of the state
    public City taskCity; The Available Task component of the state
        Only need the destination city because the source city is the current city
    public myState(City current, City next) { Constructor
        this.currentCity = current;
        this.taskCity = next;
    }
```

States: Coding

Can the state be encoded efficiently? With N cities there are N(N+1) possible states (task destination can be null for no task). Let's construct a unique index for each one and set it as the hashCode (used for java HashMap)

```
@Override
public int hashCode(){
	int taskID;
	if(this.taskCity==null){
		taskID = numCities;
	}
	else{
		taskID = this.taskCity.id;
	}
	return currentCity.id + taskID*numCities
}
```

States: Coding

HashMap also needs equals operator to deal with collisions

```
@Override
public boolean equals(Object o)
        if(o==this){
                return true;
        }
        if(o instanceof myState c){
                return this.hashCode() == c.hashCode();
        }
        else{
                Return false;
        }
}
```

States: Coding

Construct from unique hash id

```
public myState(int id) {
        int currentID = id%numCities;
        int taskID = (id-currentID)/numCities;
        this.currentCity = cityMap.get(currentID);
        this.taskCity = cityMap.get(taskID);
}
```

cityMap (line 58)

```
this.cityMap = new ArrayList<City>(this.numCities);
for(City c: topology) {
        this.cityMap.add(c.id, c);
}
this.cityMap.add(null);
```

Actions: Coding

```
private class myAction {
        The components of an action are to decide to pick up the task or not.
        If the task is picked up, the next city is determined, but if not, the next
        next city is a choice that the agent can make.
        public Boolean takeTask;
        public City nextCity;

        public myAction(Boolean take, City next) {
                this.takeTask = take;
                this.nextCity = next;
        }
```

Actions: Coding

```
@Override
public int hashCode() {
	if(this.takeTask) {
		return this.nextCity.id + numCities;
	}
	else {
		return this.nextCity.id;
	}
}

@Override
 public boolean equals(Object o) {
	if (o == this) {
		return true;
	}
	if (o instanceof myAction c) {
		return this.hashCode() == c.hashCode();
	}
	else {
		return false;
	}
 }
```

Steps

1. Construct State and Action Representations
2. Build Data Structures and Methods
3. Code Q-Learning Algorithm
4. Logist

Data Structures and Methods

public ArrayList<City> cityMap; a map from city object id to city object for convenience

public ArrayList<myState> stateList; a list of all states in id order

public HashMap<myState, ArrayList<myAction>> actionMap; a map from a state to a list of possible actions

public HashMap<myState, HashMap<myAction, ArrayList<myState>>> nextStatesMap; a map from a state-action pair to a list of possible next states

public HashMap<myState, HashMap<myAction, Double>> rewardTable; a map from a state and action pair to a reward

public HashMap<myState, HashMap<myAction, HashMap<myState, Double>>> transitionProbabilityTable; a map from current state, action, and potential future state to the probability of that transition

public HashMap<myState, Double> stateValues; a map from state to values for optimization

public HashMap<myState, myAction> policy; a map from state to optimal action

Methods: actionMap

When we observe a state we get the possible actions and store in the actionMap for more efficient future retrieval:

```
private ArrayList<myAction> getActions(myState s){
        if(this.actionMap.containsKey(s)) { efficient retrieval from memory
                return this.actionMap.get(s);
        }
        else {
                ArrayList<myAction> actions = new ArrayList<myAction>();
                if(s.taskCity!=null) { if there is a task available, we can add the pickup action
                        actions.add(new myAction(true, s.taskCity));
                }
                for(City c: s.currentCity.neighbors()) { add all move actions to neighbors
                        actions.add(new myAction(false, c));
                }
                this.actionMap.put(s, actions); store in memory
                return actions;
        }
}
```

Methods: nextStates

```
private ArrayList<myState> getNextStates(myState s, myAction a){
        if(!this.nextStatesMap.containsKey(s)) { if not in memory, initialize nested hash
                HashMap<myAction, ArrayList<myState>> actionList =
                new HashMap<myAction, ArrayList<myState>>(this.numMyActions,1.0f);
                this.nextStatesMap.put(s, actionList);
        }
        if(this.nextStatesMap.get(s).containsKey(a)) { efficient retrieval from memory
                return this.nextStatesMap.get(s).get(a);
        }
        ArrayList<myState> states = new ArrayList<myState>();
        City cCity = a.nextCity;
        myState noTask = new myState(cCity, null); construct state with no task
        states.add(noTask);
        for(City c: this.T.cities()) { add a state corresponding to a task for every possible destination city
                myState nextState = new myState(cCity, c);
                states.add(nextState);
        }
        this.nextStatesMap.get(s).put(a, states); store in memory
        return states;
}
```

Data Structures: rewardTable

```
private void buildRewardTable() {
        this.rewardTable = new HashMap<myState, HashMap<myAction, Double>>(this.numMyStates, 1.0f);
        for(myState s: this.stateList) { iterate over every possible state
                HashMap<myAction, Double> actionList =
                        new HashMap<myAction, Double>(this.numMyActions, 1.0f);
                for(myAction action: this.getActions(s)) { iterate over every possible action for that state
                        City source = s.currentCity;
                        City target = action.nextCity;
                        double cost = V.costPerKm()*source.distanceTo(target); cost of moving
                        double reward; reward for completing task if taken
                        if(action.takeTask) {
                                reward = this.td.reward(source, target);
                        }
                        else {
                                reward = 0.0;
                        }
                        actionList.put(action, reward-cost);
                }
                this.rewardTable.put(s, actionList);
        }
}
```

Data Structures: transitionProbabilityTable

```
private void buildTransitionProbabilityTable() {
        this.transitionProbabilityTable =
        new HashMap<myState, HashMap<myAction, HashMap<myState, Double>>>(this.numMyStates,
        1.0f);
        for(myState s: this.stateList) { iterate over every possible state
                HashMap<myAction, HashMap<myState, Double>> actionList =
                new HashMap<myAction, HashMap<myState, Double>>(this.numMyActions, 1.0f);
                for(myAction a: this.getActions(s)) { iterate over every possible action for that state
                        HashMap<myState, Double> nextmyStateList =
                        new HashMap<myState, Double>(this.numMyStates, 1.0f);
                        for(myState nextS: this.getNextStates(s, a)) { iterate over every possible next state
                                double probability = this.td.probability(a.nextCity, nextS.taskCity); get probability
                                nextmyStateList.put(nextS, probability);
                        }
                        actionList.put(a, nextmyStateList);
                }
                this.transitionProbabilityTable.put(s, actionList);
        }
}
```

Steps

1. Construct State and Action Representations
2. Build Data Structures and Methods
3. Code Q-Learning Algorithm
4. Logist

Q-Learning

initialize V(S) arbitrarily

loop until good enough

    loop for $s \in S$

        loop for $a \in A$

            $Q(s,a) \leftarrow R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V(s')$

        $V(s) \leftarrow max_a Q(s,a)$

Q-Learning

```
private void valueIteration(double discount, double threshold) {
    HashMap<myState, Double> myStateValueCopy =
        new HashMap<myState, Double>(this.numMyStates, 1.0f);
    for(int i=0; i<this.numMyStates; i++) {
        myState s = new myState(i);
        myStateValueCopy.put(s, this.stateValues.get(s));
    }
```

Making a deep copy of state values data structure to store temporary values

Q-Learning

```
double delta;
do {
        for(myState s: this.stateList) { we will compute the new value for each state
                double maxQvalue = Double.NEGATIVE_INFINITY;
                for(myAction a: this.getActions(s)) { compute Q value for each possible action
                        double qValue = this.rewardTable.get(s).get(a); reward term
                        for(myState nextS: this.getNextStates(s, a)) { expected future value term
                                double transProb = this.transitionProbabilityTable.get(s).get(a).get(nextS);
                                qValue+=discount*transProb*myStateValueCopy.get(nextS);
                        }
                        if(qValue>maxQvalue) { keep track of max Q-value for Value and Policy
                                maxQvalue = qValue;
                                this.policy.put(s, a);
                        }
                }
        myStateValueCopy.put(s, maxQvalue); update temporary storage with new Value
        }
```

Q-Learning

Stopping criteria based on comparing how much the state values have changed vs a threshold

```
            delta = 0.0;
            for(myState s: this.stateList) {
                double vsDelta = this.stateValues.get(s) - myStateValueCopy.get(s);
                this.stateValues.put(s, myStateValueCopy.get(s)); update Value storage with temp values
                delta+= vsDelta*vsDelta;
            }
        } while(delta>threshold);
}
```

Q-Learning: Top Level

Build Data Structures

```
this.actionMap = new HashMap<myState, ArrayList<myAction>>(this.numMyStates, 1.0f);

for(int i=0; i<this.numMyStates; i++) {
        this.stateList.add(new myState(i));
}

this.nextStatesMap =
        new HashMap<myState, HashMap<myAction,ArrayList<myState>>>(this.numMyStates, 1.0f);

this.buildRewardTable();
this.buildTransitionProbabilityTable();

this.stateValues = new HashMap<myState, Double>(this.numMyStates, 1.0f);
for(myState s: this.stateList) {
        this.stateValues.put(s, 0.0);
}
this.policy = new HashMap<myState, myAction>(this.numMyStates, 1.0f);
```

Run Algorithm

```
this.valueIteration(discount, threshold);
```

Steps

1. Construct State and Action Representations
2. Build Data Structures and Methods
3. Code Q-Learning Algorithm
4. Logist

Logist

```
public Action optimalAction(Vehicle vehicle, Task availableTask) {
        Construct the current state
        Action action;
        City currentCity = vehicle.getCurrentCity();
        City nextCity;
        if(availableTask==null) {
                nextCity = null;
        }
        else {
                nextCity = availableTask.deliveryCity;
        }
        myState currentState = new myState(currentCity, nextCity);
        myAction optimalAction = this.policy.get(currentState); Get optimal action from policy
        if(optimalAction.takeTask) { Convert to logist action
                action = new Pickup(availableTask);
        }
        else {
                action = new Move(optimalAction.nextCity);
        }
        return action;
}
```

Logist

```
private QValueIteration optimizer;

@Override
public void setup(Topology topology, TaskDistribution td, Agent agent) {
        this.optimizer = new QValueIteration(topology, td, agent);
}

@Override
public Action act(Vehicle vehicle, Task availableTask) {
        return optimizer.optimalAction(vehicle, availableTask);
}
```