# Intelligent Agent Report
# A Reactive Agent for Pickup and Delivery Problem

Team 08 Cheng-Hsiang Chiu & YiLin Hu

## Introduction

The assignment is to implement a reactive agent to solve the pickup and delivery problem by applying reinforcement algorithm for finding optimal policy.

## Implementation

Since the algorithm is reinforcement learning, an easy way to implementation is via table lookup. Firstly, we constructed a state table and a corresponding action table to model the whole problem. The state table is implemented in HashMap with current city and destination as key and existence of task in the current city as value. While the action table goes with a boolean of picking up the task as key and next city as value. Partial codes are shown in Fig.1 and 2.

Then, a reward which describing the balance between task's reward and cost based on the current state and the action is built. There is one concern here that the empty-handed agent cannot move a next city which is not a neighbor city of the current city. To prevent this scenario, the reward is penalizing with a infinite small number, say -100000000 in our implementation. The code is demonstrated in Fig 3.

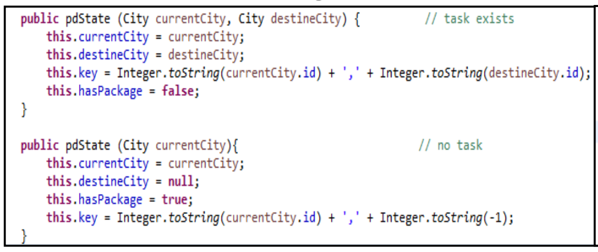
A transition is also mandatory to expression the conditional probability of a proceeding state given the current state and action. Here some situations need to be treated carefully. While the code is presented in Fig 4.

Additionally, we create best policy table and temporary table recording intermediate policy. Here we loop the value iteration 1000 times to make sure no difference between temporary policy and best policy. The code is given in Fig 5.

Finally, there are three conditions that agents could encounter when being assigned a new action. There are (1) no package in current city and moving to next city, (2) giving up of existence of package in current city and moving to next city, and (3) taking present package in current city and approaching destination. The code is denoted in Fig 6.

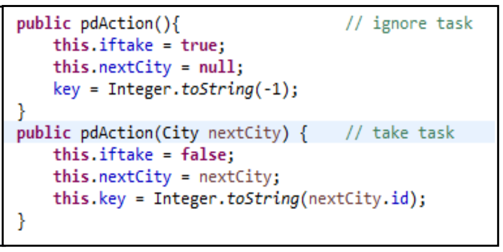
## Experimental Result

The experiment is run with one, two, and three agents respectively under nine cities described in configuration files.



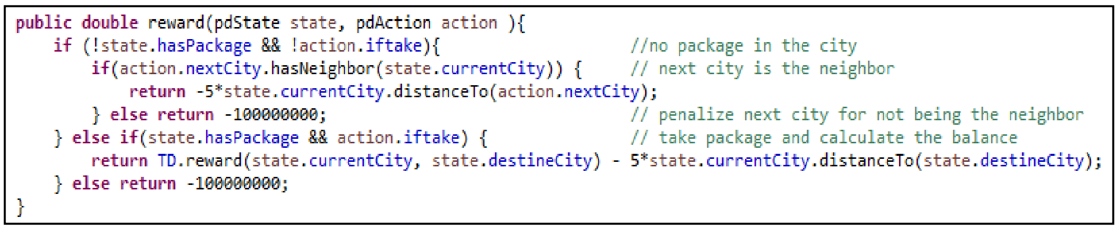
```java
public pdState (City currentCity, City destineCity) {        // task exists
    this.currentCity = currentCity;
    this.destineCity = destineCity;
    this.key = Integer.toString(currentCity.id) + ',' + Integer.toString(destineCity.id);
    this.hasPackage = false;
}

public pdState (City currentCity){                            // no task
    this.currentCity = currentCity;
    this.destineCity = null;
    this.hasPackage = true;
    this.key = Integer.toString(currentCity.id) + ',' + Integer.toString(-1);
}
```

Figure 1. Declaration of state



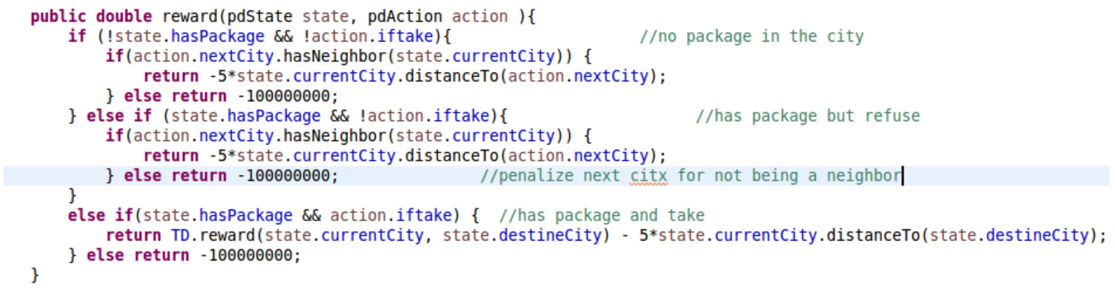
```java
public pdAction(){                          // ignore task
    this.iftake = true;
    this.nextCity = null;
    key = Integer.toString(-1);
}
public pdAction(City nextCity) {     // take task
    this.iftake = false;
    this.nextCity = nextCity;
    this.key = Integer.toString(nextCity.id);
}
```

Figure 2. Declaration of action



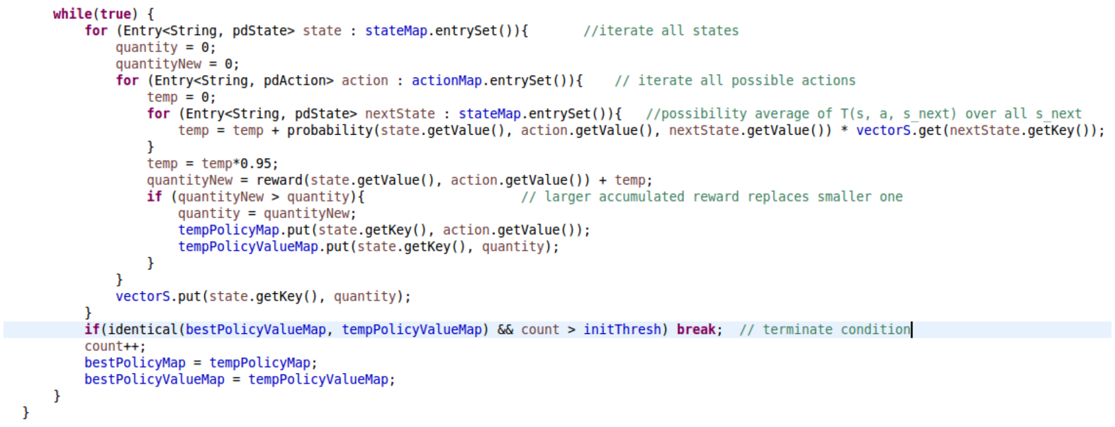
```java
public double reward(pdState state, pdAction action ){
    if (!state.hasPackage && !action.iftake){                   //no package in the city
        if(action.nextCity.hasNeighbor(state.currentCity)) {     // next city is the neighbor
            return -5*state.currentCity.distanceTo(action.nextCity);
        } else return -100000000;                               // penalize next city for not being the neighbor
    } else if(state.hasPackage && action.iftake) {              // take package and calculate the balance
        return TD.reward(state.currentCity, state.destineCity) - 5*state.currentCity.distanceTo(state.destineCity);
    } else return -100000000;
}
```

Figure 3. Implementation of reward function



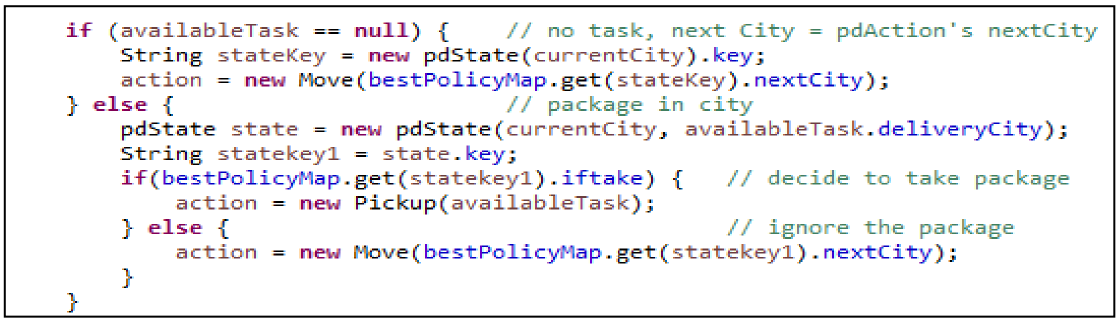
```java
public double reward(pdState state, pdAction action ){
    if (!state.hasPackage && !action.iftake){                   //no package in the city
        if(action.nextCity.hasNeighbor(state.currentCity)) {
            return -5*state.currentCity.distanceTo(action.nextCity);
        } else return -100000000;
    } else if (state.hasPackage && !action.iftake){                     //has package but refuse
        if(action.nextCity.hasNeighbor(state.currentCity)) {
            return -5*state.currentCity.distanceTo(action.nextCity);
        } else return -100000000;                   //penalize next citx for not being a neighbor
    }
    else if(state.hasPackage && action.iftake) {  //has package and take
        return TD.reward(state.currentCity, state.destineCity) - 5*state.currentCity.distanceTo(state.destineCity);
    } else return -100000000;
}
```

Figure 4. Implementation of transition function



```java
while(true) {
    for (Entry<String, pdState> state : stateMap.entrySet()){       //iterate all states
        quantity = 0;
        quantityNew = 0;
        for (Entry<String, pdAction> action : actionMap.entrySet()){    // iterate all possible actions
            temp = 0;
            for (Entry<String, pdState> nextState : stateMap.entrySet()){   //possibility average of T(s, a, s_next) over all s_next
                temp = temp + probability(state.getValue(), action.getValue(), nextState.getValue()) * vectorS.get(nextState.getKey());
            }
            temp = temp*0.95;
            quantityNew = reward(state.getValue(), action.getValue()) + temp;
            if (quantityNew > quantity){                             // larger accumulated reward replaces smaller one
                quantity = quantityNew;
                tempPolicyMap.put(state.getKey(), action.getValue());
                tempPolicyValueMap.put(state.getKey(), quantity);
            }
        }
        vectorS.put(state.getKey(), quantity);
    }
    if(identical(bestPolicyValueMap, tempPolicyValueMap) && count > initThresh) break;  // terminate condition
    count++;
    bestPolicyMap = tempPolicyMap;
    bestPolicyValueMap = tempPolicyValueMap;
}
}
```

Figure 5. Implementation of iteration



```java
if (availableTask == null) {    // no task, next City = pdAction's nextCity
    String stateKey = new pdState(currentCity).key;
    action = new Move(bestPolicyMap.get(stateKey).nextCity);
} else {                        // package in city
    pdState state = new pdState(currentCity, availableTask.deliveryCity);
    String statekey1 = state.key;
    if(bestPolicyMap.get(statekey1).iftake) {   // decide to take package
        action = new Pickup(availableTask);
    } else {                                    // ignore the package
        action = new Move(bestPolicyMap.get(statekey1).nextCity);
    }
}
```

Figure 6. Implementation of three actions

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0,-1~2 | 1,-1~2 | 2,-1~8 | 3,-1~5 | 4,-1~5 | 5,-1~4 | 6,-1~5 | 7,-1~4 | 8,-1~6 |
| 0,1~-1 | 1,0~-1 | 2,0~-1 | 3,0~4 | 4,0~5 | 5,0~-1 | 6,0~-1 | 7,0~4 | 8,0~-1 |
| 0,2~-1 | 1,2~-1 | 2,1~-1 | 3,1~-1 | 4,1~-1 | 5,1~-1 | 6,1~-1 | 7,1~-1 | 8,1~-1 |
| 0,3~-1 | 1,3~-1 | 2,3~-1 | 3,2~-1 | 4,2~-1 | 5,2~-1 | 6,2~-1 | 7,2~-1 | 8,2~-1 |
| 0,4~-1 | 1,4~-1 | 2,4~-1 | 3,4~-1 | 4,3~-1 | 5,3~-1 | 6,3~-1 | 7,3~-1 | 8,3~-1 |
| 0,5~-1 | 1,5~-1 | 2,5~-1 | 3,5~-1 | 4,5~-1 | 5,4~-1 | 6,4~-1 | 7,4~-1 | 8,4~-1 |
| 0,6~-1 | 1,6~-1 | 2,6~-1 | 3,6~-1 | 4,6~-1 | 5,6~-1 | 6,5~-1 | 7,5~-1 | 8,5~-1 |
| 0,7~2 | 1,7~-1 | 2,7~-1 | 3,7~-1 | 4,7~-1 | 5,7~-1 | 6,7~-1 | 7,6~-1 | 8,6~-1 |
| 0,8~-1 | 1,8~-1 | 2,8~-1 | 3,8~-1 | 4,8~-1 | 5,8~-1 | 6,8~-1 | 7,8~-1 | 8,7~-1 |

# Intelligent Agent Report
# A Reactive Agent for Pickup and Delivery Problem

Team 08 Cheng-Hsiang Chiu & YiLin Hu

Table 1. Final Strategy

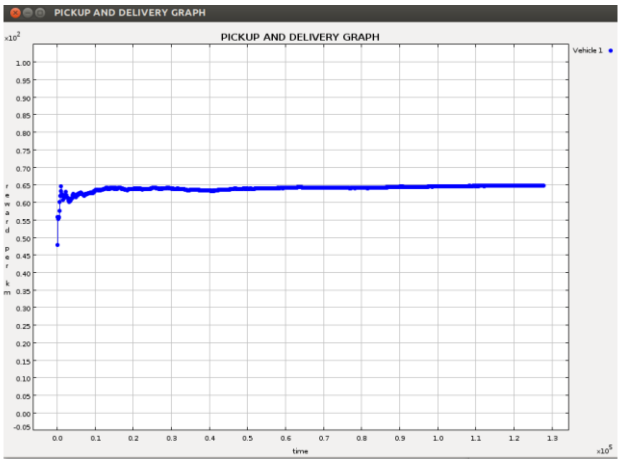


Figure 7. Simulation result in one agent

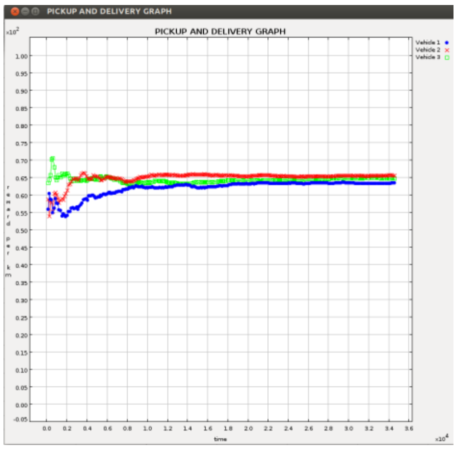


Figure 8. Simulation result in three agent

## Conclusion

The table1 is the final strategy got: (first digit represents current city, second digit represents package destination with -1 for no package, the two represent the current state key. As for the third digit, it is the action taken, with -1 represents for taking the package while others for move to next city with city id).

Figure 7 and figure 8 are simulation results for one agent and three agents respectively.

The assignment is a good practice for us to understanding reinforcement learning. The example code released on moodle achieves a 0.5 reward per km under randomly assigning a next city to empty-handed agents. While our implementation is 0.65 reward bigger than the example code, which means a bigger improvement waiting for us.