Deliberative Agent: Model Solution

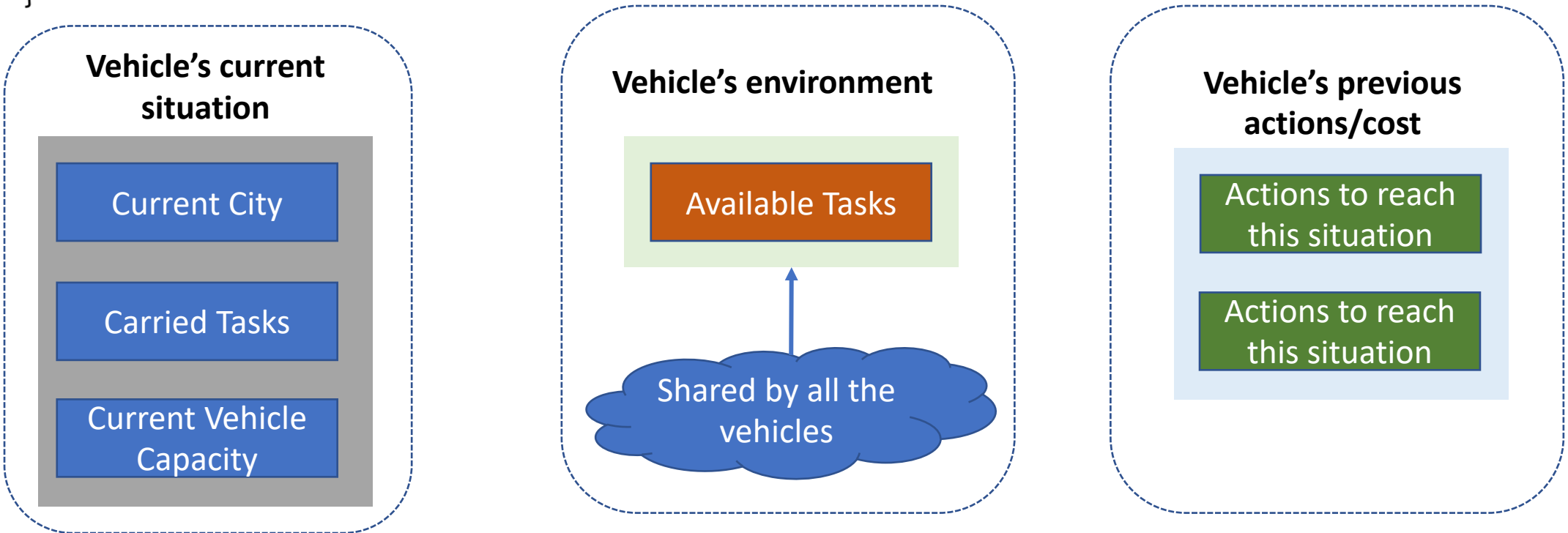
By Shaobo Cui

1. **State Definition**
    - How to define a unique state?
2. **BFS Algorithm**
    - Generate its following children node(state).
    - PickUp children and Delivery children
3. **A* Algorithm**
    - Definition of heuristic function

1. **State Definition**
    - How to define a unique state?
2. **BFS Algorithm**
    - Generate its following children node(state).
    - PickUp children and Delivery children
3. **A* Algorithm**
    - Definition of heuristic function

State Definition

```java
public State (City currentCity, TaskSet carriedTasks, TaskSet availableTasks, int currentVehicleCapacity) {
    this.currentCity = currentCity;
    this.carriedTasks = carriedTasks;
    this.availableTasks = availableTasks;
    this.currentVehicleCapacity = currentVehicleCapacity;
    this.actionsToReach = new LinkedList<Action>();
    this.costToReach = 0;
}
```



State Definition

Vehicle's current situation

- Current City
- Carried Tasks
- Current Vehicle Capacity

Vehicle's environment

- Available Tasks
- Shared by all the vehicles

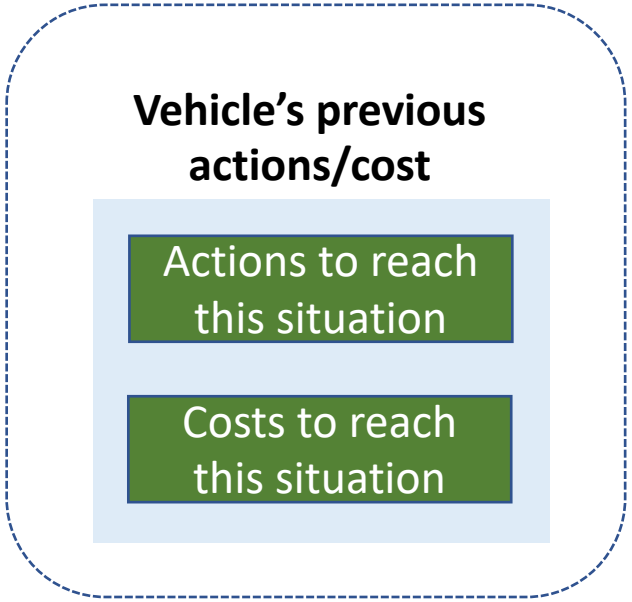

Vehicle is In which city?


Carried task?

How much capacity left?

Available tasks

State Definition: Previous Actions/Costs



- Keep record of agent's action to reach to this node, including its costs to this state.

```
public State (City currentCity, TaskSet carriedTasks, TaskSet availableTasks, int currentVehicleCapacity) {
    this.currentCity = currentCity;
    this.carriedTasks = carriedTasks;
    this.availableTasks = availableTasks;
    this.currentVehicleCapacity = currentVehicleCapacity;
    this.actionsToReach = new LinkedList<Action>();
    this.costToReach = 0;
}
```

1. State Definition
    - How to define a unique state?
2. **BFS Algorithm**
    - Generate its following children node(state).
    - PickUp children and Delivery children
3. A* Algorithm
    - Definition of heuristic function

Breadth First Search Algorithm

Think why there we use a HashMap<State, Double>

```java
public static Plan bfsPlan(Vehicle vehicle, TaskSet available, State initialState) {
    Queue<State> Q = new LinkedList<~>();
    // To store the cost for each node. If we come back to the same node, we will check this to see if we found a better path.
    HashMap<State, Double> history = new HashMap<~>();
    ArrayList<State> finalStates = new ArrayList<~>();

    Q.add(initialState);
    while (!Q.isEmpty()) {
        State node = Q.remove(); // Pop the state with the least cost
        // Check whether the state is a final one. In that case we return the plan
        if (node.isFinal())
            finalStates.add(node);
```

Advantages:

- Always finds the shortest plan.
- Not penalized by bad initial decisions.

Disadvantage:

- Requires large amounts of memory to store all nodes of the tree at each level.

Breadth First Search Algorithm

- If this is the first time to be visited: put it in history.
- If this node's cost is smaller than before, replace the with the original state.

```
    // We add the note and its children if
    // 1) this is the first time it is visited or
    // 2) the cost has decreased
    if (!history.containsKey(node) || (node.getCostToReach() < history.getOrDefault(node, Double.MAX_VALUE))) {
        history.put(node, node.getCostToReach());
        Q.addAll(node.generateChildren());
    }
}
// Return the optimal state
// we can have reached the final states multiple times. We keep the best path.
State optimalState = Collections.min(finalStates);
return new Plan(vehicle.getCurrentCity(), optimalState.getActionsToReach());
}
```

If the state is update (either added first time or updated due to the minimum cost changes), **update itself and its children**.

Generate Its Children Node(State)

1.generateDeliveryChildren():deliver one of its carried tasks

- Firstly, the agent should firstly move to the delivery city of the task.
- Then deliver the task and update its capability.
- Get the cost of children state node.

2.generatePickUpChildren():pick up one of available tasks.

- Firstly, the agent should move to the city of the task.
- Then pick up the task and update its capability.
- Get the cost of children state node.

Generate Its Children Node(State)

```java
public LinkedList<State> generateChildren(){
    LinkedList<State> children = this.generateDeliveryChildren();
    children.addAll(this.generatePickUpChildren());
    return children;
}
```

Delivery Children State: deliver one of its carried tasks

```
for (Task task: this.getCarriedTasks()) {
    // We do a deep copy
    TaskSet childCarriedTasks = this.getCarriedTasks().clone();
    childCarriedTasks.remove(task);
    TaskSet childAvailableTasks = this.getAvailableTasks().clone();

    // Create the state
    State child = new State (
            task.deliveryCity,
            childCarriedTasks,
            childAvailableTasks,
            currentVehicleCapacity: this.getCurrentVehicleCapacity() + task.weight);
```

The new state is:

1. Current location is the task's delivery city.

2. Remove the delivered task from the carried tasks.

3. Update the Vehicle's capacity.

Delivery Children State: deliver one of its carried tasks

```
// We add the action move to every city in the path
LinkedList<Action> branchActions = new LinkedList<~>(this.getActionsToReach());
for (City city: currentCity.pathTo(task.deliveryCity)) {
    branchActions.add(new Action.Move(city));
}
// And finally, the delivery action
branchActions.add(new Action.Delivery(task));
child.setActionsToReach(branchActions);

// Compute the distance
double branchCost = currentCity.distanceTo(task.deliveryCity);
child.setCostToReach(this.getCostToReach()+branchCost);

children.add(child);
}
```

1. Move to the delivery city

2. Deliver the task.

3. Compute the cost (distance) of this new state.

Pickup Children State: PickUp One of Available Tasks

```
for (Task task: this.getAvailableTasks()) {
    if (this.getCurrentVehicleCapacity() >= task.weight) {
        TaskSet childCarriedTasks = this.getCarriedTasks().clone();
        childCarriedTasks.add(task);
        TaskSet childAvailableTasks = this.getAvailableTasks().clone();
        childAvailableTasks.remove(task);

        State child = new State (
                task.pickupCity,
                childCarriedTasks,
                childAvailableTasks,
                currentVehicleCapacity: this.getCurrentVehicleCapacity() - task.weight);
```

The new state is:

1. Current location is the task's pickup city.
2. Add the pickup task to the carried tasks.
3. Remove the pickup task from the available tasks.
4. Update the Vehicle's capacity.

Pickup Children State: PickUp one of Available Tasks

1. Move to the pickup city.

2. Pickup the task.

3. Compute the cost(distance) of this new state.

```
LinkedList<Action> branchActions = new LinkedList<~>(this.getActionsToReach());
for(City city : currentCity.pathTo(task.pickupCity)) {
    branchActions.add(new Action.Move(city));
}
branchActions.add(new Action.Pickup(task));
child.setActionsToReach(branchActions);

double branchCost = currentCity.distanceTo(task.pickupCity);
child.setCostToReach(this.getCostToReach()+branchCost);

children.add(child);
}
}
```

1. **State Definition**
    - How to define a unique state?
2. **BFS Algorithm**
    - Generate its following children node(state).
    - PickUp children and Delivery children
3. **A* Algorithm**
    - Definition of heuristic function

A* Algorithm

In `PriorityQueue`, elements are retrieved in sorted order.

```java
public static class StateComparator implements Comparator<State> {
    @Override
    public int compare(State s1, State s2) { return (int) (s1.getCostFunctionValue() - s2.getCostFunctionValue()); }
}
```

```java
public static Plan aStarPlan(Vehicle vehicle, TaskSet available, State initialState) {
    // Efficient data-structure to get cities that have the lowest score
    PriorityQueue<State> Q = new PriorityQueue<~>(new StateComparator());
    // To store the cost for each node. If we come back to the same node, we will check this to see if we found a better path.
    HashMap<State, Double> history = new HashMap<~>();

    Q.add(initialState);
```

A* Algorithm

Always pop out the state with the least cost.

But how to estimate the cost of State?

```
    while (!Q.isEmpty()) {
        State node = Q.remove(); // Pop the state with the least cost

        // Check whether the state is a final one. In that case we return the plan
        if (node.isFinal()) {
            return new Plan(vehicle.getCurrentCity(), node.getActionsToReach());
        }

        // We add the note and its children if 1) this is the first time it is visited or 2) the cost has decreased
        if (!history.containsKey(node) || (node.getCostToReach() < history.getOrDefault(node, Double.MAX_VALUE))) {
            history.put(node, node.getCostToReach());
            Q.addAll(node.generateChildren());
        }
    }

    return null; // We should never reach this step
}
```

A* Algorithm: Cost Function Estimation

```
public double getCostFunctionValue () {
    return this.getCostToReach() + this.getHeuristicValue();
}
```

$$f(n) = g(n) + h(n) \ ,$$

But how to get heuristic value of state.

There are many possible ways as long as the heuristic value is the **underestimate** of the true value.

A* Algorithm: Cost Function Estimation

1. Get all the cities that the agents have to visit.
2. Get all the cities that the agents already visited.
3. Two loops to iterate through all visited ones.
    a. For a ToVisitCity, we suppose that we could always locate in this city's nearest visited neighbors.
    b. Get this minimum value
    c. Add this ToVisitCity from ToVisitCity set to the VisitedCity set.

This is an obvious underestimate as we can't always ensure that we can reach to the ToVisitCity from its nearest neighbor. In other words, the current city is not always the nearest neighbor.

A* Algorithm: Cost Function Estimation

```
public double getHeuristicValue () {
    double minCostBetweenCities = 0;
    Set<City> citiesVisited = new HashSet<~>();
    Set<City> citiesToVisit = new HashSet<~>();

    // Get all the cities that we will visit
    for (Task task: this.getCarriedTasks())
        citiesToVisit.add(task.deliveryCity);
    for (Task task: this.getAvailableTasks()) {
        citiesToVisit.add(task.pickupCity);
        citiesToVisit.add(task.deliveryCity);
    }

    // We remove our current city
    citiesToVisit.remove(this.getCurrentCity());
    citiesVisited.add(this.getCurrentCity());
```

- Build the `citiesToVisit` and `citiesVisited` set.
- The final state is that the `citiesToVisit` set is empty.

How to estimate the heuristic value?

A* Algorithm: Cost Function Estimation

```
    // Go through all cities to compute the heuristic
    while (!citiesToVisit.isEmpty()) {
        double minDistance = Double.MAX_VALUE;
        City closestCity = null;

        for (City cityVisited: citiesVisited) {
            for (City cityToVisit: citiesToVisit) {
                double distance = cityVisited.distanceTo(cityToVisit);
                if (distance < minDistance) {
                    minDistance = distance;
                    closestCity = cityToVisit;
                }
            }
        }

        citiesToVisit.remove(closestCity);
        citiesVisited.add(closestCity);
        minCostBetweenCities += minDistance;
    }

    return minCostBetweenCities;
}
```

Iterate two loops to obtain the shortest way to reach to certain city.

Thanks!