Excercise 3
Implementing a deliberative Agent

Group №2: Cosmin-Ionut Rusu, Sorin-Sebastian Mircea

October 22, 2019

1 Model Description

The state must be designed in a way in which it will uniquely identify all the possible scenarios the world can be in. Too few information kept as a state and we will be missing out on optimization opportunities and even in finding a solution (where all the tasks are delivered).

State

(currentCity, delivering, toDeliver, totalAgentCapacity) .

Where *delivering* represents the tasks the agent is delivering at the current time (picked up) while the *toDeliver* are tasks waiting to be picked up. From *totalAgentCapacity* (value that doesn't change during a run) and *delivering* we can infer the current capacity that the agent possesses.

1.1 Intermediate States

Are represented by all the states that are not final, where the agent has still tasks to pick-up or deliver.

$$toDeliver.size() > 0 \tag{1}$$

or

$$delivering.size() > 0 \tag{2}$$

1.2 Goal State

A goal state is characterised by the length of the *delivering* and *toDeliver* tasks HashMap being zero.

1.3 Actions

In this order, in both A* and DFS algorithm we are doing the following actions:

1. Deliver picked-up task in the current city (if possible)
2. Check if the state is final
3. Take available tasks from current city (going to take one by one and add it to the state); there are cases in which there are more tasks than the agent can take at a given time, so which action we choose could influence the results
4. Move to one of the neighbors of the current city

2 Implementation

The two objects that hold the tasks (*delivering* and *toDeliver*) are implemented as a *HashMap of Tasks*

As our BFS and A* work directly with Plans, whenever we fork a given plan and take an action, we deep copy that object and place it in the Queue / PriorityQueue along with the state.

2.1 BFS

A classical BFS on a graph with uneven weights will not necessarily find the first state to also be the most optimal one (it will render the shortest path in terms of nodes traveled, but the cost associated to that road can be higher than other paths that consist in more nodes traveled).

Implemented with a Queue and a HashMap that maps states to costs (helping us avoid cycles). Whenever we are transitioning to a new state we look in the HashMap to be sure that we either haven't explored that state or we have optimized the cost that it takes to reach it.

2.2 A*

It is very similar to the BFS, the differences consist in using a PriorityQueue instead of a Queue and taking into consideration the Heuristic when choosing to explore or not a newly generated State.

The way in which we are checking if we need to explore a newly generated state is the following:

```
Long prospectiveEarn = (newPlan.totalDistanceUnits() + computeHeuristic(newState));
prospectiveEarn *= costPerKm;
if(!statesMap.containsKey(newState) || ( prospectiveEarn < statesMap.get(newState)))
    statesMap.put(newState, newPlan.totalDistanceUnits() * costPerKm);
    pq.add(new QueueParams(newState, newPlan));
}
```

The PriorityQue comparator function

```
Comparator<QueueParams> qpComparator = (p1, p2) -> (int) (
    (p1.getPlan().totalDistanceUnits() + computeHeuristic(p1.getState()))
    * costPerKm
    -
    (p2.getPlan().totalDistanceUnits() + computeHeuristic(p2.getState()))
    * costPerKm
);
```

2.3 Heuristic Function

Our estimating cost is lower than the true cost as the agent will surely need to do extra kilometers to and from pickup / delivering cities.

```
private Long computeHeuristic(State state) {
    long cost = 0;

    for(Task task: state.getDelivering()) {
        cost += task.pickupCity.distanceTo(task.deliveryCity);
    }
    for(Task task: state.getToDeliver()) {
        cost += task.pickupCity.distanceTo(task.deliveryCity);
    }

        return cost;
}
```

3 Results

3.1 Experiment 1: BFS, A* and Naive Agent Comparison

3.1.1 Setting

We are using Switzerland's topology, the number of tasks is 6 with a random seed of 23456

3.1.2 Observations

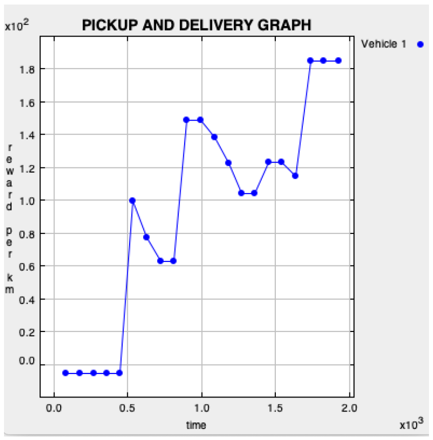

Figure 1: A*; totalDistanceUnits plan of 1380000

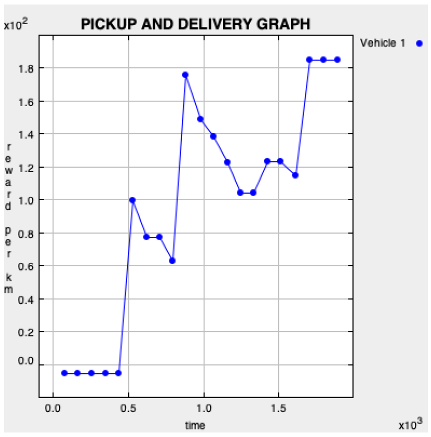

Figure 2: BFS; totalDistanceUnits plan of 1380000

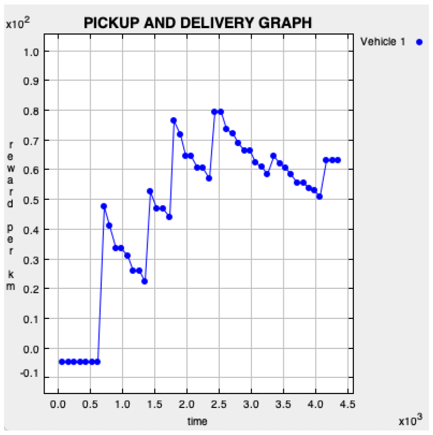

Figure 3: Naive; totalDistanceUnits plan of 3840000

The way that the BFS and A* are designed, both return the optimal solution. Our version of the BFS has a theoretical complexity of $O(n * m)$ while the A* has complexity of $O(m * \log_2 n)$ but the BFS is faster in practice. Within a time frame of 1 minute, the BFS can solve a maximum of 11 tasks, while the A* can solve 10.

3.2 Experiment 2: Multi-agent Experiments

3.2.1 Setting

We are using Switzerland's topology, the number of tasks is 10 with a random seed of 23456

3.2.2 Observations

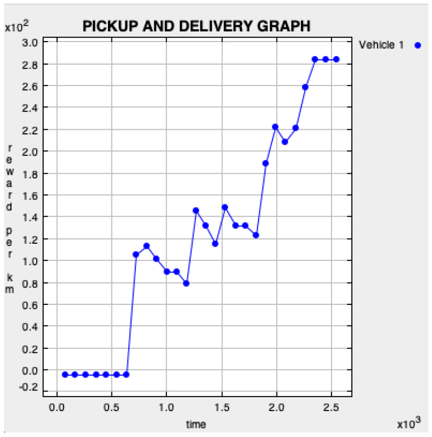

Figure 4: 1 agent; 1 plan computation

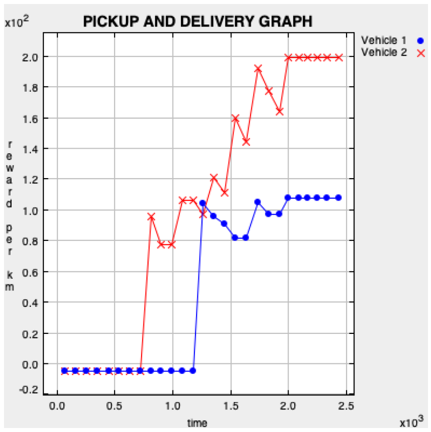

Figure 5: 2 agents: 6 plan computations

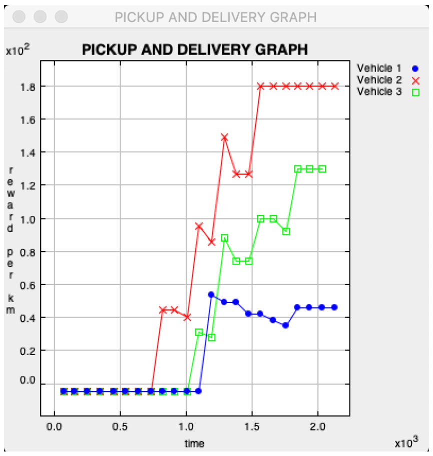

Figure 6: 3 agents: 13 plan computations

We can observe that the number of plan re-computations is doubling as we increase the number of agents. This is a signal that the optimal plans we are computing for each agent clash and are rendered invalid. In theory, this leads to losses (as we are to increase the number of total agents in the world) but during our tests, the world and the number of tasks are big enough to accommodate the larger amount of agents and to results in an overall better total reward per km (for all the agents)