Excercise 3
Implementing a deliberative Agent

Group48 №: Boubacar Camara, Ivan Snozzi

October 20, 2020

1 Model Description

We decided to represent the state as a tuple of three elements: the city where the agent is at the moment, a set containing the tasks we picked (and still have to deliver) and a set containing the task which are still available to pick.

$$State := (currentCity, picked, toPick) \tag{1}$$

When we move from a city to another, the *currentCity* changes. When we pick up a task we add it to the set *picked* and remove it from the set *toPick*. Upon delivery, the task is removed from *picked*.

1.1 Intermediate States

The intermediate states are states where at least one of the sets *picked* or *toPick* is non empty. If the set *toPick*$\neq \emptyset$ it means that there are still available tasks to pick up on the map. If the set *picked*$\neq \emptyset$ it means that we still have some tasks to deliver.

1.2 Goal State

A goal state is reached when *picked*$=\emptyset$ and *toPick*$=\emptyset$. In other words an end goal is reached when we picked up and delivered all the task on the map.

1.3 Actions

There are two actions: **PICKUP** and **DELIVER** :

- **PICKUP:** moves the agent from *currentCity* to a city target which has an available task to pick up and picks it up.
- **DELIVER:** moves the agent from *currentCity* to the delivery destination of the task and delivers it.

2 Implementation

States are represented by the **State** class which contains the current city, the sets of picked tasks and tasks to pick as described in the previous section. To model the graph, we added the class **Node** which contains a state, a reference to its parent Node, the cost to reach it from the root node (passing by the referenced parent) and the list of actions its parent state should execute to reach it.

A node does not contain the list of it's child nodes. Instead, these children can be generated dynamically by calling the function **computeNeighbors(Node parent, double parentCost)** which returns the List of child Nodes. Each node in this list is initizialized with the correct state, costs, and list of previous actions to reach it.

Both BFS and A* star algorithms use similar data structures. We represented, the queue of nodes to visit **Q** by a **LinkedList<Node>**. The data structure **C** used to store visited nodes is modeled by a **HashMap<State, Node>** where the key is simply the node state. The map is convenient in order to update the parent node, cost and list of previous actions of a node visited more than once.

The concrete implementation of the two algorithms is close to the pseudocode given as reference. Implementation details for both algorithms are explained below.

2.1 BFS

To build the best plan, we need to find the shortest path from the root to one of the leaves. In order to do this using BFS, we modified the algorithm to update the queue **Q**, and the map **C** when a shorter path to an already visited node is found. The node with the corresponding state in **C** is updated with the new cost, parent and list of previous actions. The cost of children nodes in **Q** are recomputed to take the new cost into account. Once all nodes have been visited, we backtrack from the leaf with the lowest cost up to the root in order to aggregate the actions required the build the best plan.

2.2 A*

Using A* algorithm, the queue **Q** is sorted at the end of each iteration and we return the first leaf reached. We added a new field fcost to the Node class which represents the cost computed by adding the heuristic cost to the initial one.

2.3 Heuristic Function

We used two heuristics for the A* algorithms.

2.3.1 hMaxDistance(node)

This heuristic computes the maximum distance to one of the cities that must be visited from the current city. We must visit all delivery cities from the picked up tasks and all pickup cities from the resting tasks to pick. We can compute the distance of the shortest path from the current city to any of these cities. Since we must go to the furthest of these cities, we are sure that the optimal path must go through this city. The cost of the optimal path is at least the cost of the shortest path to the furthest city. Hence this heuristic preserves optimality of A*.

2.3.2 hAverage(node)

This heuristic is the distance (in kilometers) from the current city to the center of mass of the cities we must visit. The idea behind this average is to force the agent to discover states corresponding to cities that are close to the ones it must visit. Given the same argument, this heuristic gives a lower result than the previous one, since the average distance to cities to visit is lower than the maximum distance described in the previous section. Hence hAverage preserves optimality as well.

3 Results

3.1 Experiment 1: BFS and A* Comparison

3.1.1 Setting

Topology: Switzerland, Seed: 23456.
The experiments were done on a Intel Core i7-8650U CPU @ 1.90GHz × 8.

3.1.2 Observations

BFS	**1 task**	**2 taks**	**3 taks**	**4 taks**	**5 taks**	**6 taks**	**7 taks**	**8 taks**	**9 taks**
Time [ms]	4	6	11	28	73	197	1403	21865	timeout
Distance	610	890	890	1220	1220	1380	1610	1710	-

A*	**1 task**	**2 taks**	**3 taks**	**4 taks**	**5 taks**	**6 taks**	**7 taks**	**8 taks**	**9 taks**
Time [ms]	2	4	4	7	27	129	713	12816	290215
Distance	610	890	890	1220	1220	1380	1610	1710	1729

From the two tables above we can see how the heuristic function helps the algorithm to converge faster to the optimal solution. For the **hMaxDistance** heuristic function the plan computing time is on average half of the BFS computing plan time. In both cases the traveled distance is the same, which was expected, since they should both converge to the same solution.
The maximal number of tasks for which we can build a plan under one minute is 8.

3.2 Experiment 2: Multi-agent Experiments

3.2.1 Setting

Topology: Switzerland, Tasks: 6, Agents: deliberative-bfs, Seed: 23456, Cost per km: 5

3.2.2 Observations

In the multi agent setting, agents compete to pickup tasks but they are not aware of each other. A plan is invalid when a task planned to be picked has already be taken by another agent. This results in the recomputation of the plan at the pickup city. However the recomputed plan is not guaranteed to optimal due to the presence of competitors. In our experiment with two agents, they both need to recompute their plan once around 1000 ticks and the second agent named vehicle 2 does an unnecessary travel around 1200 tick in order to pick a task already taken by the first agent. If the second agent considered the opponent when building its plan, it would not have planned to pick up this last task.

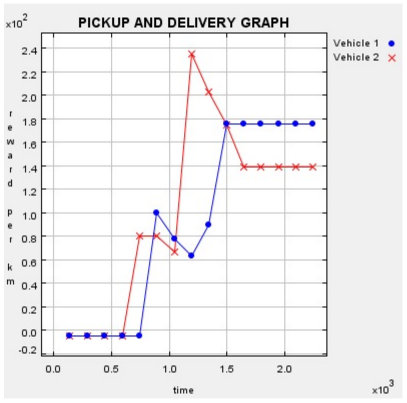

Figure 1: Plot of the reward per km in a setting with two agents and 6 tasks.