Exercise 3
Implementing a deliberative Agent

Group №22: Dubuis Samuel, Facklam Olivér

October 20, 2020

1 Model Description

1.1 Intermediate States

We defined our state representation through three main parameters: the agent's current location, the set of available tasks throughout the map and finally the set of tasks that were picked up. This representation defines precisely every state that the agent can find itself in, and should help it make a decision for its next action.

1.2 Goal State

A state is considered a goal state (or final state) when its picked up task set and available task set are both empty, meaning all tasks have been correctly delivered. The two algorithms BFS and A*, presented in the next section, search through the states until they get to some possible goal state(s), at which point they choose the best one.

1.3 Actions

In our model, the agent has three possible actions :

1. Move : If the agent decides to move, the successor state will have the same sets of available and picked up tasks but the location will be different and a cost of moving will have been added.
2. Pickup : In the case of picking up a task, the location will not change in the successor state, however the said task will be removed from the available tasks and will be added to the picked up set.
3. Deliver : In the case of delivering a task, the location will not change but this time the task will be removed completely from the picked up tasks set. The set of available tasks remains the same.

2 Implementation

In this section we explain how we implemented particularities in our code.

The first main step was to code the `State` class, implementing the representation described in the previous section.

We then created a `Node` class. This is a wrapper around `State`, which – in addition to the state itself – also contains a pointer to the previous node as well as the action which led to this state. This allows to easily reconstruct the optimal plan when a goal node is found.

The `Node` class has for parameters the `parentNode`, the `previousAction`, the `state`, the `cost` and finally the `vehicle`. It also implements a few methods useful for our agent implementation. In particular, it

implements the function `public List<Node> getSuccessors()` that returns, as its name suggests, the list of successor `Node`s, used mainly to compute next actions and continue exploring the state space. Also, it has a heuristic method that we will explain in detail in a further subsection.

A separate `Algorithms` class was created, which contains both algorithms. Finally, it is worth mentioning that we create the plan for our agent by basing ourselves on the best final `Node`. We move backward in our chain of actions by iterating and return a final plan of actions.

2.1 BFS

First we defined the BFS algorithm. We started by simply implementing the basic algorithm, however this one returns the first final `Node` it finds. We changed it so that if a state is considered final and its cost is less than the current `bestNode`, it replaces it. At the very end we return the best `Node` to compute our plan from.

2.2 A*

The A* algorithm doesn't change much from the basic one seen in class. It is worth noting that we used a `PriorityQueue` to optimize the sorting of the list Q. `Node`s in the queue are sorted based on their cost, and this cost is computed through a heuristic function giving the estimated future cost of getting to a final state. As a reminder, states can be considered multiple times if they have a lower cost than previous encounters of the same state.

2.3 Heuristic Function

Our main idea for the heuristic function is that the future cost will always be more than the cost for picking up and/or delivering the most expensive available task. Thus we implemented this function in our `Node` class. We loop over all available tasks of the current state and compute their cost (going to pickup and then to delivery), keeping the maximum one. We repeat this with the set of picked up tasks, considering their cost to go to delivery. Finally we return `maxCost`, the highest cost among the considered tasks.

Since any plan leading to a goal state needs to include the pickup and/or delivery of this most expensive task, the real future cost is higher than the `maxCost` returned by the heuristic. Thus the heuristic underestimates the actual cost: the heuristic is admissible. This in turn guarantees the optimality of the A* algorithm.

3 Results

3.1 Experiment 1: BFS and A* Comparison

3.1.1 Setting

We set the random seed to the default 23456 so that all agent have the same tasks. We also used the basic topology predefined in the default settings, which is the Swiss topology.

3.1.2 Observations

As we can see in the table 1, the BFS algorithm takes a lot more time to setup than the A* one. This is obviously the result we were looking for. It is also worth noting that depending on computer specs, results as the two below can differ.

Both algorithms achieve the same final cost and the same final profit, since both of them are optimal.

Number of tasks	6	9	12
BFS Computation time	0.098s	1.137s	81.317s
A* Computation time	0.047s	0.721s	30.506s
BFS total profit	254657	459169	652649
A* total profit	254657	459169	652649
BFS total cost	6900	8600	9100
A* total cost	6900	8600	9100

Table 1: Different results for both algorithm on the same setup, for different number of tasks

On one of our computers, we could create a plan for 11 tasks under one minute with the BFS algorithm, and 13 tasks under one minute for the A* algorithm. For more than 14 tasks we would get timeout errors.

3.2 Experiment 2: Multi-agent Experiments

3.2.1 Setting

This experiment was also executed on the Swiss topology, with a random seed of 123. Throughout the different simulations, we varied the number of tasks as well as the number of agents. We simulated systems with 1, 2 and 3 agents, acting independently on the same map containing 6, 9 or 12 tasks. The agents operate with the A* algorithm, since it produces the same result in a shorter time.

3.2.2 Observations

Total cost	1 agent	2 agents	3 agents
6 tasks	3400	7500	9600
9 tasks	5800	11000	13950
12 tasks	7000	12050	16000

Table 2: Total cost with multiple agents on map

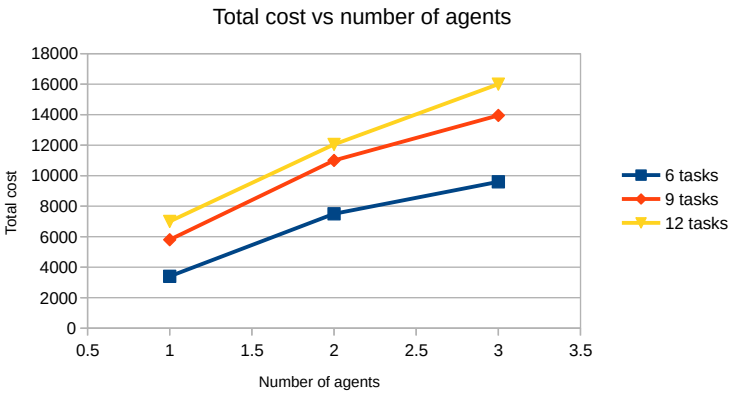

Figure 1: Joint cost of multi-agent system

As expected, the total joint cost of a multi-agent system increases with the number of agents. This is because each agent operates independently, calculating an optimal plan to deliver all tasks, ignoring the other agents' existence. Since all the agents are trying the deliver the same tasks, so-called collisions happen: one agent gets the task, while the others have travelled for nothing.

Obviously, the total cost also increases with the number of available tasks, since more tasks generally incur more travel distance. However it can be noted that the slope of this total cost (as visible on figure 1) is roughly the same for all tested amounts of tasks. It seems that the cost increase created by adding more agents to the system does not really depend on the number of tasks. A probable hypothesis is that this cost increase mainly depends on the topology.