Excercise 3
Implementing a deliberative Agent

Group №41: Mohammadreza Ebrahimi, Salar Rahimi

October 20, 2020

1 Model Description

At first, we are aware of the initial state since they are imposed by the initial settings of our files (*.xml* files). In the following we describe how we defined our states.

1.1 Intermediate States

We included all the necessary attributes that affect the states. First, we consider the *currentCity* since all the following nodes will need this attribute. Then, we needed *vehicle* so that we can infer the capacity and location of the agent. Indeed, *totalCost* is essential since we base our optimality in that sense. We also have *load* which indicates how much in weight we are carrying. The *Plan* that the agent follows is also included in the state representation and that's because that will be the output of our algorithms ($A*$ and *BFS*). Finally, we need both *remainingTasks* and *carryingTasks* because in the case that they are both empty, it means we are at leaf (possible goal) state. Furthermore, we overriden the *equals* method so that we declare what is our reasoning for two states to be equivalent. This method compares two states based on their *currentCity*, *remainingTasks* and *carryingTasks*. We handle the *totalCost* difference in different manners between two different plans.

1.2 Goal State

According to the above explanations, the goal state is defined as one of the possible leaf nodes meaning that both *remainingTasks* and *carryingTasks* are empty array lists. For *BFS*, we keep track of all such leaf nodes and choose the optimal one on the sense of *load* and in the case of $A*$, the first leaf node we encounter is our goal state.

1.3 Actions

We decided to define possible action as *Pickup* and *Move* actions. The chief reason is that it is clearly mentioned in the lab description that once we get to a delivery city, we drop the package without (in this case,Implemented in Code) "thinking". Therefore, the number of successors for each node will be the number of neighbors plus one. To briefly explain, we can mention that moving to each state, updates *currentCity*, *totalCost* and *Plan*. It might be the case that in neighbor cities we can deliver packages which in turn updates *remainingTasks*, *carryingTasks*, and *load*. Since, dropping packages has zero cost, we drop all the possible packages in the neighbor city (we don't consider dropping only one package and maybe another one to limit the state space). To explain the extra one state, we can say it's the outcome of the action *Pickup* from *currentCity*. Also, in this case we fetch all the packages in this city and also consider the capacity of our vehicle. Conceivably, this action updates *remainingTasks*, *carryingTasks*, *load* and *Plan*.

2 Implementation

2.1 BFS

Since we described our state representation and action/transition properly, defining this plan became relatively straightforward. As it was mentioned before, we handled the equivalence of different states uniquely. For the case of *BFS*, if the current node is formerly visited and the current *totalCost* is lower, we eliminate the former node and all of its successors (otherwise it becomes computationally extremely immense). Doing so, drops the computation time significantly. However, the *BFS* algorithm is by far inferior in terms of speed of convergence comparing to $A*$. Final essential remark is that we don't stop the algorithm when we find the first leaf node and instead, we continue to search for all the possible leaf nodes and pick the one(s) with minimal *totalCost*. This measure ensures the optimally of the BFS as it explored all the possible nodes and find the one with the least cost.

2.2 A*

For this algorithm, we defined an extra parameter *fn* that will substitute for *totalCost* and accoring to the algorithm description it's the sum of *totalCost* and the heuristic function (denoted by *predCost* in our code). We handle equivalent states based on the value function that we defined. This value function is naturally based on the heuristic function that will be explained in the next section.

2.3 Heuristic Function

Heuristic function provides an extra information about the goal. It is an underestimate of the cost to reach the goal for any given state. Therefore, it is of great importance to define an function that would not overestimate the problem. Wrong heuristic may lead the algorithm toward the local minimum (which actually happened for us when we didn't have a minimal heuristic).

For any state to reach the goal, it requires to perform all the remaining tasks as well as delivering the carrying ones. We decided to consider a gain for each of the these lists. However, because the carried task are already picked up we considered half penalty compare to the remaining tasks. The heuristic function is as follows:

$$h(n) = R_t.P_r + D_t.P_d$$

Where R_t is *remainingTasks*, P_r is the penalty for them, D_t is *deliveryingTasks*, P_d is the penalty of them. The heuristic ensures the optimality of A* as it contains enough information about the environment and does not push the agent toward one local min and rather guides it to the optimal solution.

3 Results

3.1 Experiment 1: BFS and A* Comparison

3.1.1 Setting

To compare our plans and determine optimality, we examine both algorithms multiple times. We started with two tasks and increased this number to 7 for both algorithms. However the final costs were identical for both which proves the optimality of the heuristic function.

3.1.2 Observations

In Table 1, we have provided a comparison between two algorithms in terms of *totalCost* and the consumed time for each to compute and execute the plan by one agent. Also, we realized that within one minute, we can handle 6 tasks with *BFS* and 8 tasks with A* (rngSeed = 23456). We tested on other random

Table 1: Comparison of the cost and time for both algorithms

Algorithm	NTasks	Total Cost	Time(s)	Algorithm	Total Cost	Time(s)
BFS	2	4450	0.013	A*	4450	0.007
BFS	3	4450	0.025	A*	4450	0.012
BFS	5	6100	1.269	A*	6100	0.078
BFS	6	6900	17.94	A*	6900	0.595

seed and similar performance was observed. In overall, A* algorithm is way faster as it has the heuristic and total cost as a guidance to open new nodes.

3.2 Experiment 2: Two-agent Experiment

3.2.1 Setting

For both algorithms, we set the number of tasks to 6 and we let the agents to update their plan and respective cost. With the default seed number, one of the agents start from Lausanne and the other one from Zurich.

3.2.2 Observations

At the beginning the *totalCost* is 6900 (as before) for agent1 and 5950 for agent 2. Since we have adversaries, the plans should be recomputed and executed. As the change of plan (which is computationally much faster), agent one yields a *totalCost* of 4300 and this value is 3250 for the second agent. The same plan and re-computation happened for both algorithms.

3.3 Experiment 3: Three-agent Experiment

3.3.1 Setting

For both algorithms, we set the number of tasks to 7 and we let the agents to update their plan and respective cost. Agent1 starts at Lausanne, Agent2 at Zurich and Agent3 at Bern.

3.3.2 Observations

In the following, we bring the *totalCost* estimation of each agent:

$\text{Agent}_{N}umber$	$1st$ estimate	$2nd$ estimate	$3rd$ estimate
1	8050	6450	7100
2	3565	6100	5650
3	2700	4300	-

Table 2: Cost estimation and update for each agent

3.4 Conclusion for Multiple Agents

Lack of communication between the agents has caused agents to be blind regarding of their plan, and this lead to multiple re-planing whenever the selected plan fails. The expense of such a design in the case of multiple agents is computational time spend for multiple re-planing. For large number of tasks the problem would be more challenging.