Excercise 3
Implementing a deliberative Agent

Group №16: Léopold Bouraux, Vincent Rinaldi

October 22, 2019

1 Model Description

1.1 Intermediate States

For an agent, states are defined by 6 different attributes. First of all we have the *currentCity* which is the current location of the agent, the *remainingCapacity* which is the additional weight the vehicle can still carry, and the *currentCost* which is computed by multiplying the total distance traveled by the vehicle by its *costPerKm* parameter. In addition, there are also two TaskSets which are *notPickedTasks* and *deliveringTasks*, being respectively the set of tasks that have not yet been picked up, and the set of tasks that are currently being delivered. Finally, *actions*, is a list of actions which gathers in chronological order all the actions that led the agent to this state. Concerning the States, *States* used in the BFS algorithm cannot be sorted, while the (A*) algorithm sort them following their *heuristic cost* : we call them *ComparableStates* in this case.

1.2 Goal State

The agent has to deliver all tasks. This happens when the lists of Tasks *notPickedTasks* and *deliveringTasks* become both empty at the same time. Since they are attributes of a State, if those lists are empty for a given State, then that means this State is goal state.

1.3 Actions

In order to choose which transition will be made (which action will be taken) we iterate on our two sets of tasks and for each set we have two possible scenarios. The first scenario is, for each available task not yet picked up, and if the task weight is less than the remaining cost allowed to transport, if the agent's current position is the same as the pickup City of the task, then a next possible state is created in which a PickUp action is added to its action list, and the current task is added to its tasks to be delivered and removed from the its not picked up tasks set (we also don't forget to subtract the remaining capacity of the vehicle by the weight of the new task picked up). Otherwise, if the location is not the same, it means we're on our way to go to pickup a new task and we then add a Move action to the actions list of the next State such that the destination city of the Move action is the the next city on the path between our current position and the pickup location of the task. Then we also update the next state's current city, and we add the vehicle travel cost penalty to the next state cost. The second scenario is, for all tasks that are still being delivered, we have two options. The first one is if our current position is the same as the delivery destination of the currently evaluated task, then we must deliver it by creating a new State, adding a Deliver action to its list of actions, removing this task from its delivering tasks set, and increasing the remaining capacity of the vehicle by the task weight. Otherwise the second option comes, and since it means that we are on our way to deliver the task to its destination, we then create a new state by updating its cost of the trip to the next city, adding a Move action to the next city to its actions list, and updating its next current city.

2 Implementation

2.1 BFS

To start to algorithm, we first initialize a starting state that we add to a standard *Linked List* queue. While the queue is not empty, at each iteration of our *while* loop, we pop the first element from the queue and firstly check if it is a final state. If it's the case, then we can terminate the execution of the BFS. Otherwise, we check if we didn't previously discovered this State. If not, we add the current evaluated state to the set of the discovered ones. Then we compute all the possible next states as described above. After creating the corresponding State for each existing task, we add each of those states to the queue, and repeat the procedure until we find a final state. Once it is found, the action list allows us to compute the agent's route from the initial State to the goal one.

2.2 A*

The A* algorithm is almost the same as the BFS, with the only difference that the queue will not be a standard *Linked List* but a *Priority Queue* of *ComparableState*. This comparator will override the method *compare* and will give the ability to our *Priority Queue* to sort in ascending order the States by their heuristic cost. This means that the next State to compute on in the queue will not be chosen arbitrarily anymore but will follow a priority condition for optimality.

2.3 Heuristic Function

Our heuristic function can be represented with the formula $f(n) = g(n) + h(n)$ where n is our state, $g(n)$ is the current cost of the state, and $h(n)$ is the highest possible additional cost we can add to the current travel cost when being in this State. This means that we select the task, either if we still have to pick it up and deliver it or if we just have to deliver it, that increases the cost of travel by the highest possible amount among all remaining tasks for this State. The idea behind this heuristic in terms of optimality is that we want to avoid overestimating our model or the true final cost. Indeed, by always giving computation priority to the States proposing the least costly "furthest task" to process (either to pick it up and deliver it, or simply deliver it), we will mostly avoid overestimating the true final cost regarding the other tasks that are still pending. Taking into account the future possible states instead of always choosing immediately the least costly task for a given State, brings us closer to optimality on the long term, but also gives us an adequate computation time since the States in our queue are not anymore added in arbitrarily order.

3 Results

3.1 Experiment 1: BFS and A* Comparison

3.1.1 Setting

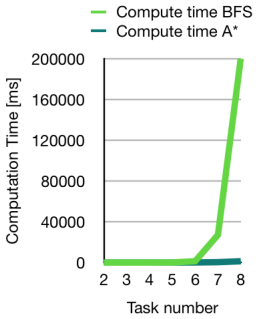
We ran our two algortihms using a seed equal to 23456, and a number of agents ranging from 2 to 11 in order to compare their performance.

3.1.2 Observations

First of all we can see that the number of tasks for which we can build a plan in less than one minute with BFS is 7. For 8 tasks, it takes more than 3 minutes and timeouts for more than 8, whereas with A*, we can build a plan up to 10 tasks in less than one minute, and it still doesn't timeout for more than 10. Secondly, regardless of the number of tasks, A* produces fewer iterations than BFS and performs approximately the same for a small number of tasks, but is drastically better than BFS, especially in terms of execution time, when having 5 or more tasks to deliver around the world map. We can indeed see that the computation time for the

Nb of tasks:		2	3	4	5	6	7	8	9	10	11
BFS	# iterations	62	312	1761	6962	26956	104708	382908	-	-	-
	Compute tm	8ms	11ms	31ms	132ms	1321ms	27206ms	200283ms	-	-	-
	Total profit	73222	132445	190018	208466	254657	322824	389802	-	-	-
	Total cost	4450	4450	6100	6100	6900	8150	8550	-	-	-
A*	# iterations	55	187	1262	4134	18447	86586	337345	1139172	4084299	13467718
	Compute tm	14ms	19ms	32ms	60ms	119ms	415ms	1336ms	3773ms	13498ms	91495ms
	Total profit	73222	132445	190018	208466	254657	322924	389802	459169	528086	583232
	Total cost	4450	4450	6100	6100	6900	8050	8550	8600	9100	9100

Comparison of the number of iterations, computation time and optimality of the result for both algorithms



Computation time evolution in function of the number of tasks

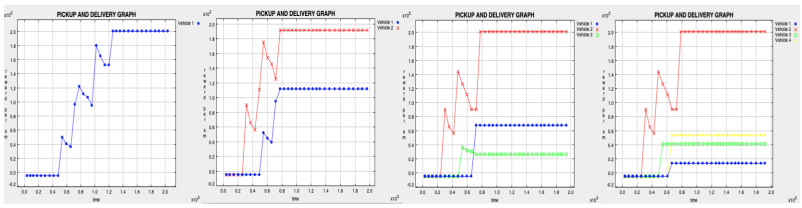
BFS grows exponentially faster than for A*. The heuristic we chose to build A* then produced a more optimal computation to our problem.

3.2 Experiment 2: Multi-agent Experiments

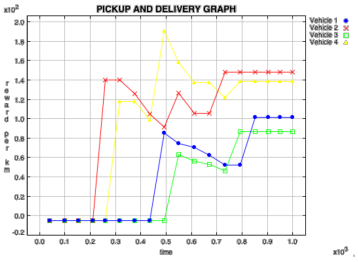
3.2.1 Setting

For this experiment, we kept the same seed and topology. We created 1, then 2, then 3, then 4 agents at the same time with A* and 7 tasks, then 10 tasks, to handle.

3.2.2 Observations



(a) Evolution of the reward/km for each (1, 2, 3, 4) vehicle(s), 7 tasks, using A*



(b) Evolution of the reward/km for each 4 vehicles, 10 tasks, using A*

We notice that the more vehicles there are, the faster the final state is reached. We can see that agent 2 replaces agent 1 in the second simulation, resulting in a decrease in agent 1's remuneration in the second case. Agent 1 could take tasks from agent 2, so agent 2 gets into a final state faster since less tasks are remaining. The same reasoning can be applied to the other two experiments. Since tasks tend to be taken over and performed by an other agent, the agents simply stop considering the tasks at this moment and consider it has been delivered to keep efficiency. However, in this kind of case, one agent seems to maximize its personal reward/km way more than the others for a small number of tasks. If we increase the total number of tasks in the world map, like with 10 tasks, we can see that, the work is much better distributed among them. In terms of computation time, we recorded for each vehicle (respectively for Vehicle 1, 2, 3, 4 with 10 tasks), 13883ms, 9876ms, 2853ms and 1555ms. These computation times decrease as agents have to recompute their plan when tasks are picked up by other agents. The average reward for an agent may then also decrease when operating with other agents in parallel, comparing with the case where he is alone in the world map, but it can reach its final state more quickly since it shares the tasks with the other agents. To concluded, a higher number of agents running in parallel at the same time does not necessarily keep efficiency in terms of average profit, and we then need to find the most optimized appropriate number of agents according to the number of tasks spread in the world map that we need to handle.