Excercise 3
Implementing a deliberative Agent

Group №: 33, Gregoire Clement and Maxime Delisle

October 23, 2018

1 Model Description

In order to implement a deliberative agents, we need to have `State`s, transitions from one state to the other which are represented by `Action`s and finally one or multiple goal `State`s which are states where all tasks of the agents are done.

1.1 Intermediate `States`

Each state is represented by the following properties:

- The position of the agent (represented by a `City` class)
- The set of tasks loaded for delivery (represented by a `TaskSet` class)
- The set of tasks that needs to be picked-up for delivery (represented by a `TaskSet` class)

These are the elements used to distinct a `State` from others. In our implementation we also added more parameters such as capacity or history (past `Action`s) in order to ease the implementation of the algorithm. One should be we aware that a `State.equals(anotherState)` just means that the remaining work for the agent is the same for both `State`s but not that the past `Action`s were the same and hence the respective cost of the previous `Action`s.

1.2 Goal `State`

A goal `State`, or a final `State`, is a `State` where all loaded tasks have been delivered and there are no remaining task to be picked-up. Hence, the `State` is final if and only if both `TaskSet`s are empty.

1.3 `Actions`

Three different type of `Action`s are considered:

- `MoveAction(City c)`: The agent moves to c, a neighboring `City`.
- `DeliveryAction(Task t)`: The agent delivers t, a `Task` the agent holds, to its current location.
- `PickupAction(Task t)`: The agent picks-up t, a `Task`, at its current location.

`Action`s involve cost. For this deliberative agent, only the `MoveAction(c)` results in a cost (that our agent tries to minimize) which is `currentCitiy.distanceTo(c)` × `costPerKM`.

2 Implementation

In order to implement both BFS and A*, we added into `State`s a way to keep track of previous `Action`s and the corresponding cost of these.

Also, we created a data structure which is made of a `Queue` of `State`s and a `Set` of visited `States`. It handles like a regular `Queue` except that it can set `State`s as "visited". This changes how `State`s are added to the queue, now only non-visited `State`s are added except [∗] if it has a smaller cost (= better path for the same resulting `State`). This enables us to prevent cycles.

Finally, in order to make the numbers of `State`s we go through as small as possible, the `Action`s available from one `State` is as minimal as needed. This means, we only propose `MoveAction` to cities that are on the path of a `Task` to pickup or to deliver but not otherwise. Also, when delivering is possible, this will be the only `Action` proposed, as this could never worsen a plan.

2.1 BFS

In the case of BFS, we explore `State`s in the order that we see them. Hence we use a FIFO `Queue` (implement as a `LinkedList`) and for each `State`s we go through we add all `nextStates` which are all the states reachable corresponding to the available `Action`s to the `Queue`. We also want to find all possible final `State`s, this means we will only stop when all paths have been tried and only the best plan will be kept and returned.

2.2 A*

In the case of A*, we explore the `State` in a special (ascending) order given by `cost(s) + h(s)`, the former being the current cost of the path up to this `State s`, the latter being the heuristic function onto this `State s`. One should note that [∗] would only be necessary in the case of DFS because A* with a correct (= keeps optimality) heuristic will always see `State`s in order (regarding the cost) and hence will stop directly when it reaches a final `State`. Though [∗] makes the queue smaller and results in better performance in the case of A*.

2.3 Heuristic Function

We have tried several heuristic functions and have kept in this report the two that were best, the first which keeps optimality while significantly reducing the number of states traversed compared to the zero heuristic (which results in Dijkstra's algorithm), and the second which is crazy fast because it ends up being really greedy.

The first one (`DISTANCE_REMAINING2` in the code) adds the cost of going to the `pickupCity` of one remaining `Task`s to pickup and to the `deliverCity` of one remaining `Task`s to deliver. It only adds the best path between going to one city and then the other or the opposite, and it would only add the path to one city if one or the other does not exist (for example, no tasks to pickup left).

The second one (`WEIGHT_NOT_TAKEN` in the code) is super greedy. It forces the agent to pickup all available tasks because it penalizes states where the number or remaining `Task`s to pickup is greater. Obviously this heuristic does not guarantee the optimality of the final state but shows how heuristics can be tweak to optimize both speed and correctness of the solution.

3 Results

3.1 Experiment 1: BFS and A* Comparison

The aim of this experiment is to compare the performance and the limitation of the different agents when they are alone. We focus on an agent using BFS, an agent using A* with an optimal heuristic and second agent using A* with a faster heuristic.

3.1.1 Setting

The agents' performances are tested for [8, 10, 11, 20, 50, 100, 150] tasks on the topology of Switzerland.

3.1.2 Observations

	Random	BFS			A* fast			A* optimal		
#tasks	distance	distance	time	#states	distance	time	#states	distance	time	#states
8	4420 km	1710 km	$< 1s$	0.23 Mio	1760 km	$< 1s$	$< 1,000$	1710 km	$< 1s$	81,000
10	5180 km	1820 km	7s	3.11 Mio	1860 km	$< 1s$	$< 1,000$	1820 km	6s	0.85 Mio
11	5550 km	1820 km	34s	10.35 Mio	1860 km	$< 1s$	$< 1,000$	1820 km	20s	2.40 Mio
12	5990 km	-	-	-	1970 km	$< 1s$	$< 1,000$	1820 km	50s	6.12 Mio
20	9960 km	-	-	-	3150 km	$< 1s$	$1,100$	-	-	-
50	23760 km	-	-	-	5690 km	$< 1s$	$2,000$	-	-	-
100	44260 km	-	-	-	11870 km	$< 1s$	$5,500$	-	-	-
150	65990 km	-	-	-	18130 km	$< 1s$	$13,500$	-	-	-

As can be seen above, the plans our agents made are between 2.6 to 4 times shorter than those from the naive agent. The A* with an optimal heuristic is more efficient than BFS as it can handle one more task (12 tasks vs 11 tasks) when there is a limit of 1 minute. Moreover, A* is more flexible because, depending on the heuristic used, it can handle 150 tasks, or even more, while giving results almost optimal. (The same conclusions were reached from other topology)

3.2 Experiment 2: Multi-agent Experiments

In this experiment, we aim to show the impact of having multiple agents working at the same time, as well as the difference between BFS and A* in that regard, if there is one. Note that we will test for A* with an optimal heuristic.

3.2.1 Setting

The experiment is made on the Swiss topology. For both BFS and A*, 10 tasks were to be delivered by [1, 2, 3, 4] actors of the same kind.

3.2.2 Observations

We found that having more agents meant having a smaller reward/km ratio, which makes sense, as the agents are made to work alone and "steal" each others. We found no differences between BFS and A*, which also makes sense, as they both plan optimal routes. However, with the exception of one agent, having more agents didn't mean having to remake plans more frequently which is surprising as we would expect that having would lead to having more plans cancelled. See below for the exact values.

	BFS		A*	
#agents	ratio reward/km	#plan changes	ratio reward/km	#plan changes
1	290	0	290	0
2	160	7	175	7
3	160	5	150	8
4	135	6	135	7