Excercise 3
Implementing a deliberative Agent

Group №: 14; Iorgulescu Calin, Launay Clement

October 22, 2017

1 Model Description

1.1 Intermediate States

The state representation we chose is very simple and consists of the current city our agent is located in, and the T status of the T tasks it has to accomplish. These status are ternary variables whose three possible values are NOT_PICKED, $HOLDING$ and $DELIVERED$. Indeed, since a task can only be dropped at the delivery city once it has been picked up, these suffice to know the position of the tasks at any time (NOT_PICKED:the task is at pickupCity, $HOLDING$:it is at the agent's currentCity $DELIVERED$:it is at deliveryCity). $state = (currentCity, status)$ where $currentCity \in \{city\}$, $status \in \{progress\}^T$, T is the number of tasks and $progress \in \{NOT\_PICKED, HOLDING, DELIVERED\}$.

1.2 Goal State

Given our state representation presented above, it can easily be verified if a state is a final state. It is the case if all the T $progress$ variables evaluates to $DELIVERED$.
$IsFinal(state) = IsFinal((currentCity, status)) \iff \forall t \in status, t = DELIVERED$. In practice we will keep track of the number of task that evaluate to $DELIVERED$.

1.3 Actions

In a state $(currentCity, status)$, the available actions are given by the T values of $status$: if a task t is labeled such that $status[t] = NOT\_PICKED$, we can $PICKUP(t)$ (subject to the weights of the tasks being currently carried). If it is $HOLDING$ we can $DELIVER(t)$. So in any state, the number of possible action is bounded by T. Note that in practice, our modeling of $PICKUP(t)$ and $DELIVER(t)$ include the actions of actually moving from $currentCity$ to $t.deliveryCity$ using the shortest path available and the actual delivery action. As an example, $DELIVER(0)$ in state $(city4, \{HOLDING, DELIVERED\})$ could induce the sequence of action: $Move(1), Move(0), Delivery(0)$ (assuming the shortest path from city4 to deliveryCity(0)=city0 is city4,city1,city0). Obviously if we chose the action of delivering a task whose delivery city is the agent's current city, then it does not need to move.
This choice of transitions for our model results from the fact that any optimal plan can be described by a sequence of $PICKUP(t)$ and $DELIVERY(t)$ as we described them. Indeed, any move action of an optimal plan actually aims at going to a given city in order to perform a pickup or delivery for a task, which is what we sum up with the actions we defined. In term of transitions, executing $PICKUP(t)$ in state $s = (currentCity, status)$ means that we will transition to a state s' where $currentCity = t.pickupCity$ and $status[t]$ has changed from NOT_PICKED to $HOLDING$ (similarly $currentCity$ becomes $t.deliveryCity$ and $status[t]$ changes from $HOLDING$ to $DELIVERED$ in the case of $DELIVER(t)$. As mentioned, a $PICKUP(t)$ action is only available in a state s if $t.weight + \sum_{status[t']=HOLDING} t'.weight < vehicle.capacity$ which further reduce the amount of possible actions for a certain state. This also yields an upper bound on the number of states. Since our represen-

tation implies that exactly $2T$ actions should be executed to go from the initial state to a final state (one pickup and one delivery for each of the T tasks) and at most T actions are possible in a given state, we will have at most T^{2T} cases to consider.

2 Implementation

Our implementation relies on a class *State* which forms the nodes used in the BFS and A* algorithms. It encapsulates the representation of a state we described with two variables City inCity (the current city) and int[] taskStatus. In addition, the class also provides some shortcuts *weightCarried* and *delivered* (which avoid recomputing these results every time) and a link to the *State* which precedes it. Finally, it implements the methods required to be processed by the algorithms: $successors()$ and $f()$ (the distance function).

2.1 BFS

The implementation of the Breadth-First Search algorithm is pretty straight-forward considering how we coded *State*. We start with an initial state where all tasks are NOT_PICKED and call $successors()$ to further populate the queue. The algorithm returns when all possible final state have been evaluated (the queue is empty) and the best one is returned. Besides its disadvantages compared to A*, this algorithm also process multiple times some states reached by different path, an issue that proved not to be addressable because by the time a new path is found for node, its successors have already been queued.

2.2 A*

The implementation of A* follows the one provided in the exercise presentation. The main technical aspects are delegated to native Java objects with the use of a TreeSet¡State¿ with a custom Comparator on the distance function $f() = costToReach + heuristic$ for the queue and a HashMap¡State,Double¿ to store the set of visited nodes and the best distance found to reach them yet (our heuristic is not consistent so we have to check if we can improve it).

2.3 Heuristic Function

In order for A* to be optimal (it always returns the optimal plan), it is necessary and sufficient for our heuristic to be admissible, which means that calling $state.computeHeuristic()$ should always return a smaller value than the actual best possible cost to reach a final state from this starting state. The heuristic we chose is: $h((currentCity, status)) = max($
$max_{t=NOT\_PICKED} currentCity.distanceTo(t.pickupCity) + t.pickupCity.distanceTo(t.deliveryCity),$
$max_{t=HOLDING} currentCity.distanceTo(t.deliveryCity))$
This heuristic basically consider the cost to deliver each task individually (as if there was only this one to take care of) which is easy to compute, and take the maximum on all tasks. This can be represented as transporting the longest task on a straight path with all other tasks being picked up and delivered on the way. As a consequence, this heuristic is often far from the actual result yet it is seems non-trivial and is clearly admissible which is the main criteria.
Another idea for the heuristic would have been to solve the Traveling Salesman Problem on the set of cities that must be reached given the tasks status (the cities where a task have yet to picked up or deliver). However an heuristic based on this idea is admissible only if the solution is exact which we assumed is too much complexity to be useful.

3 Results

3.1 Experiment 1: BFS and A* Comparison

3.1.1 Setting

We run both A* and BFS (one after the other) in the exact same configuration (the given default one, same rngSeed:23456).

3.1.2 Observations

As expected, both algorithms always return the same plan (which is checked to be optimal for 2 and 3 tasks). A* is more efficient time and memory-wise and can thus handle a larger number of tasks before passing the one minute mark.

Number of tasks	Time (ms)		Number of states evaluated		Optimal cost	Reference (naive plan)
	A*	BFS	A*	BFS		
3	16	12	36	271	890.0	1590.0
5	32	13183	441	326011	1220.0	3090.0
8	395	-	44650	-	1710.0	4420.0
11	9646	-	1879264	-	1820.0	5550.0

A* is clearly superior to BFS. Its main advantage is the reduced number of states it evaluates (BFS tries all) which allow it to handle situations with up to 11 tasks, while BFS fails at 6. Remark: in these cases the optimal cost being way lower than the reference is mostly due to the tasks' weights being low (3) compared to the vehicle's capacity (30), giving our agent a tremendous advantage since it carries many tasks at once.

3.2 Experiment 2: Multi-agent Experiments

3.2.1 Setting

We reproduce the same configurations as before (default configuration, again with the same rngSeed) and simulate 3 agents each time. The agent uses ASTAR (does not actually impact the results since BFS also returns the optimal plan).

3.2.2 Observations

As we could have predicted from the *planCancelled* method we implemented (which only consists in passing the carried tasks to the next initial state), our agent do not coordinate with each other. AS a result the joint performance is much worse than what a single agent could do, as can be seen in the following array reporting the distance travelled by the agents:

number of tasks	Agent 1	Agent 2	Agent 3	Total	1 agent	naive agent
3	770.0	570.0	620.0	1960	890.0	1590.0
5	1220.0	810.0	1070.0	3100.0	1220.0	3090.0
8	990.0	1270.0	990.0	3250.0	1710.0	4420.0

It turns out that multiple deliberative agents can actually perform worst than a naive agent when there are only a few task (all agents rush the same tasks to begin with). Hopefully they tend to be better when the number of available tasks increase.