Excercise 3
Implementing a deliberative Agent

Group №: 14; Iorgulescu Calin, Launay Clement

October 24, 2017

1 Model Description

1.1 Intermediate States

The state consists of the current city our agent is located in, and the statuses of the T tasks it can deliver. A task's status is a ternary variable with the value of `NOT_PICKED`, `HOLDING`, or `DELIVERED`. Since a task can only be dropped off at the delivery city once it has been picked up, it is possible to infer the position of a given task 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

We consider a state to be final if all its task statuses are set to `DELIVERED`. In practice we keep track of the number of delivered tasks to speed up checking if a state is final.

$IsFinal(state) = IsFinal((currentCity, status)) \iff \forall t \in status, t = \texttt{DELIVERED}$.

1.3 Actions

For a given state $(currentCity, status)$, the possible actions are given by the T values of $status$. If, for a given task t, $status[t] =$ `NOT_PICKED`, we can $PICKUP(t)$ (subject to the vehicle capacity and the carried tasks' weights). If it is `HOLDING` we can $DELIVER(t)$. So in any state, the number of possible actions is bounded by T. Note that in practice, our modeling of $PICKUP(t)$ and $DELIVER(t)$ also includes the $MOVE(city)$ actions to get from $currentCity$ to $t.deliveryCity$ using the shortest path. For example, $DELIVER(0)$ in state $(city4, \{$`HOLDING,DELIVERED`$\})$ might require the following sequence of actions: $MOVE(1), MOVE(0), DELIVER(0)$ (assuming the shortest path from city4 to deliveryCity(0) is city4 - city1 - deliveryCity(0)). Obviously if we chose to deliver a task whose delivery city is the agent's current city, then there is no need to move.

This choice of transitions for our model results from the fact that any optimal plan can be described as a sequence of $PICKUP(t)$ and $DELIVERY(t)$ actions (as discussed above). Any optimal plan will entail the vehicle moving on the shortest path between cities to pickup or deliver a ask. Further, executing $PICKUP(t)$ in a state $s = (currentCity, status)$ means transitioning 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 possible in a state s if

$$(t.weight + \sum_{status[t']=HOLDING} t'.weight) < vehicle.capacity$$

which constrains the possible actions for a certain state.

This also yields an upper bound on the number of states. Since exactly $2T$ actions are 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, there will be at most T^{2T} cases to consider.

2 Implementation

The `State` class represents the nodes used in the BFS and A* algorithms. It encapsulates two variables: `City inCity` (the current city) and `int[] taskStatus`. Task statuses are encoded as integer values from 0 to 2. In addition, we also keep the current vehicle weight (`weightCarried`), the number of delivered tasks (`delivered`) (to avoid recomputing them each time), and a reference to the parent `State`. We also keep the current cost needed to reach the `State` starting from the vehicle's initial position.

The `State` class also implements methods required by the planning algorithms: `successors()` and `f()`. The former returns a list of child `State` objects, and the latter returns the distance estimation used in A*.

The equality relationship of two `State` objects is redefined to be the equality of `inCity` and of the elements of `taskStatus[]`. In keeping with Java language specifications, we also redefined the `hashCode()` method, ensuring that only these fields are considered.

2.1 BFS

We start from an initial state where all tasks are `NOT_PICKED`, calling `successors()` to further populate the queue. The algorithm retains the best final state found (the one with the lowest cost), and returns when all possible final states have been explored and the queue is empty. Alternatively, it is possible to stop after the first final state is encountered, but this planning will not necessarily be optimal. An additional possible optimization is to not expand nodes already processed, but this is infeasible when searching for the optimal solution: any time a lower-cost state is found, the queue needs to be scanned for any children and all need to have their `costToReach` fields updated. All children not already part of the queue would be re-enqueued.

2.2 A*

The implementation of A* follows the example provided in the exercise slides. We use a `TreeSet<State>` as a priority queue, with a custom `Comparator` to define ordering. This ensures that the queue is ordered by the distance function $f() = costToReach + heuristic$. A `HashMap<State, Double>` stores the set of visited nodes and the lowest cost to reach them.

2.3 Heuristic Function

In order for A* to be always return the optimal plan, it is necessary and sufficient for our heuristic to be admissible, which means that calling $state.computeHeuristic()$ should always return a value smaller than or equal to the actual best possible cost to reach a final state from this starting state. The heuristic we chose is:

$h((city, status)) =$

$$max_t \begin{cases} dist(city, t.pickupCity) + dist(t.pickupCity, t.deliveryCity) & status[t] = \texttt{NOT\_PICKED} \\ dist(city, t.deliveryCity) & status[t] = \texttt{HOLDING} \end{cases}$$

This heuristic considers the maximum cost of delivering any given task individually. The insight is that the vehicle always needs to pickup and/or deliver the task farthest from its current position. Therefore, the best case scenario can be thought of as transporting this farthest task on the shortest path with all the other tasks being picked up and delivered on the way. Given a state, there can never be a plan to handle the remaining tasks that doesn't require moving on the shortest path to the farthest tasks. As a consequence, though this heuristic is often far from the optimal, it is admissible, which is the main criteria.

It is possible to reduce this problem to the Traveling Salesman Problem, which is NP-complete: the remaining pickup/delivery cities would be the nodes. However that would defeat the purpose of the heuristic, since computing the solution would be prohibitively expensive, whereas the heuristic needs to be cheap to compute.

3 Results

3.1 Experiment 1: BFS and A* Comparison

3.1.1 Setup

We run both A* and BFS (one after the other) using 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 since the first final state it finds is guaranteed to be optimal, 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 since BFS needs to explore all states. Given the reduced number of states A* explores, it can 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 Setup

We use 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 updates the set of carried tasks for the next initial planner `State`), our agents do not coordinate with each other. As a result, the joint performance is much worse than for a single agent. This can be seen in the following array reporting the distance traveled 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 single naive agent when there are only a few tasks, since all agents will rush to the same tasks in the beginning. Hopefully they tend to be better when the number of available tasks increase.