Excercise 3
Implementing a deliberative Agent

Group №46: Yann Vonlanthen, Loïc Vandenberghe

October 22, 2019

1 Model Description

1.1 Intermediate States

A state is represented by 3 attributes: $s = (location, carried, tasks)$ where *location* is the city where the vehicle is. *carried* is the set of tasks that are carried by the vehicle and *tasks* is the set of tasks that are available in the world.
Note that $s \in S$, where $S \subseteq Cities \times \mathcal{P}[Tasks] \times \mathcal{P}[Tasks]$, and $\mathcal{P}$ denotes the power-set. S is a proper subset of $Cities \times \mathcal{P}[Tasks] \times \mathcal{P}[Tasks]$ iff the set of tasks $Tasks$ is not empty, as a task can't both be carried and available. However the number of states still grows exponentially with respect to $|T|$. This state representation is needed in order to provide optimal planning.

1.2 Goal State

A state is defined as a goal state if $carried = \emptyset$ and $tasks = \emptyset$. Note that the location doesn't matter, as the agent doesn't care in which city he terminates.

1.3 Actions

From any intermediate state $s = (location, carried, tasks)$ the possibles actions and their resulting states are:

- $\forall c \in neighbors(location)$: **Move to city** c**.** $\Rightarrow (c, carried, tasks)$
- $\forall t \in carried \wedge (deliveryCity(t) = location)$: **Deliver** t**.** $\Rightarrow (location, carried \setminus \{t\}, tasks)$
- $\forall t \in tasks \wedge (pickupCity(t) = location) \wedge (weight(t) + weightSum(carried) \leq capacity)$: **Pickup** t**.** $\Rightarrow (location, carried \cup \{t\}, tasks \setminus \{t\})$

2 Implementation

2.1 BFS

In order to keep track of the path from the initial state to any state we added 3 attributes to the class `State` in addition to *location*, *carried* and *tasks*. The first attribute is called *parent* and contains a pointer to the state that precedes this state. Then we have added 2 attributes called *pickup* and *delivered*, who represent the single task that was either picked-up or delivered to arrive at this resulting state. These attributes do not define the class, i.e. two states are still considered equal if the 3 base attributes *location*, *carried* and *tasks* are equal.
The methods hashCode() and equals() have been redefined to match this definition of a state.

The BFS initial state is the current city of the vehicle, with the last carried tasks (that we update in plan cancelled) and the available tasks. Further we define two helper methods: `computeSuccessors( State s)` creates a list of all states that can be reached by applying all possible actions to s, and `constructPlanFromGoal( State goal)` which starts from a goal state and constructs a path by going back through parents until reaching the initial state.
Finally we also assumed the following: If at a given state, the vehicle can deliver a task, `computeSuccessors( State s)` will return a single state where one task is delivered. Since there is no reason for the task to be carried, and as the order of delivery in the same city does not impact the cost of the path, we can slightly reduce the complexity of the algorithm by selecting a single task to deliver and not consider other actions yet.

In order to achieve **optimality**, we do not return the first goal state found. However, we disregard any further state that has a higher cost than the current best goal state, which significantly improves performance.

2.2 A*

For this algorithm, we added the interface Comparable to `State` and use a Min-Heap and the respective Priority queue implementation in Java to make sure that the queue is ordered efficiently.
States are compared based on the evaluation function $f(s) = g(s) + h(s)$. $g(s)$ is the cost from the start state to the current state and $h(s)$ is a heuristic estimating the cost from the current state to a goal state, described in 2.3.
To compute $g(s)$ we store the cost of the last action in `State` and add recursively the cost of the parent states. The only action that costs something is moving between two cities, the cost being the number of kilometers traveled. (i.e. we don't take the positive rewards of delivering a task into account because at the end all the tasks need to be delivered anyways, and ordering isn't important)

2.3 Heuristic Function

The main idea of our heuristic function is to take the maximum cost needed to complete one of the remaining tasks as estimation. In other words $h(s)$ returns the maximum number of kilometers left for the vehicle to deliver a single task when taking the shortest path. More formally for a state s we define:

$$h(s) = \max\left[\max_{t \in carried} dist(location, delivery(t)), \max_{t' \in tasks} dist(location, pickup(t')) + dist(pickup(t'), delivery(t'))\right]$$

For A* to be optimal we need to prove that $h(s)$ is *admissible*, i.e. that it's always optimistic compared to the effective cost needed to reach a goal state. (This leads to an optimal result as no goal state with potentially lower costs will have been overlooked when the first goal state is reached.)

The proof of admissibility follows immediately from the fact that a goal state is defined by having $carried = \emptyset$ and $tasks = \emptyset$. Thus any task will eventually have to be completed, and by taking the minimum cost needed to do so, we have $h(s) \leq h^*(s)\ \forall s \in S$, where $h^*(s)$ is the optimal cost to reach a goal state from s. Note that we take the maximum over all minimum costs in order to have a tighter bound, which will speed up the informed search. Note also that this heuristic can be stored after being computed once and thus its complexity of $\mathcal{O}(|T|)$ is not detrimental to performance.
Tighter heuristics exist, but we chose to use this one as it is linear in the number of tasks and seemed like a good trade-off.

3 Results

3.1 Experiment 1: BFS and A* Comparison

3.1.1 Setting

We used the Swiss topology provided in Switzerland.xml. In addition, we ran the experiment with multiple numbers of tasks for each of the following three planning strategies: BFS, A* basic which means that it's heuristic is always 0, and A* with the heuristic described in sec 2.3.

3.1.2 Observations

Topology	Number of tasks	Shortest path length	Computation Time		
			BFS	A* basic	A*
Switzerland	8	1710km	1.20 s	0.59 s	0.18 s
	10	1820km	8.99 s	4.22 s	1.17 s
	12	1820km	>2min	90.76 s	5.13 s
	14	1820km	>2min	>2min	13.41 s
	15	1820km	>2min	>2min	39.8 s

Since every algorithm gives the optimal path, the choice of the algorithm does not influence the cost of the solution. However, we can see that, at some point, the number of tasks saturate the topology. Which means that even by adding more tasks, the optimal solution cost does not increase much. This is because, with more tasks, it becomes easier to transport and complete multiple tasks within the same path. And since the vehicle is already going to every nodes for 10 tasks, if the capacity allows to carry 5 additional tasks, the agent can use the same path for 15 tasks.

In terms of efficiency, A* with a good heuristic can compute the optimal solution for 15 tasks in the Swiss topology in less than 1 minute. BFS can only compute 11 tasks in less than 1 minute.

Since the complexity is exponential, the time to compute the solution grows quickly with the number of tasks. By choosing a bad heuristic, the A* algorithm is still faster than BFS because A* does not have to search through all the possibles goal states, as unlike BFS, A* search firsts for the lower cost states before considering the long one. A* with a good heuristic is far faster than both other algorithms since it avoids computing costly paths before having tried cheaper ones.

3.2 Experiment 2: Multi-agent Experiments

3.2.1 Setting

For this experiment, we used the Swiss topology with 10 tasks available tasks, and compare the performance of 1,2 and 3 vehicles.

3.2.2 Observations

Number of vehicles	Vehicle 1		Vehicle 2		Vehicle 3		Total
	Path	Tasks	Path	Tasks	Path	Tasks	KM
1	1820 km	10	-	-	-	-	1820
2	1180 km	5	1110 km	5	-	-	2290
3	970 km	4	1410 km	4	1180 km	2	3560

When two vehicles are operated, we only see a slight increase in distance covered (roughly a factor of 1.25), but the tasks are delivered around 1.5 times quicker. This means that two vehicles operate quite well together in this setting. However, for three vehicles the total distance covered increases rapidly, and even worse, more time passes until the last task is delivered than with two. In short, we observe a trade-off between a quick delivery and an efficient one, but with too many vehicle neither is achieved.