Exercise 3
Implementing a deliberative Agent

Group №:44 Valentin Kindschi, Cyril van Schreven

December 2, 2019

1 Model Description

1.1 Intermediate States

A state is characterized by three variables: a city, the tasks left and the task currently carried. The tasks are saved in a TaskSet logist objects. The states are implemented in the *DeliberativeSate* class, which also contains fields to store its total cost, travel cost and heuristic cost. These costs are specific to the context in which this state has been attained. To be able to compare states according to the correct variables, the 'equals' function has been overriden and redefined.

1.2 Goal State

The goal state, or final state as we called it, is a state without any task left and without any task carried. The class *DeliberativePlan* has a function called $searchFinalState()$ that finds the easiest goal state before generating its full sequence of actions of the plan.

1.3 Actions

An action is defined by the action of either picking up or dropping a package, and a corresponding task. This behavior is implemented in the *DeliberativeAction* class.

2 Implementation

2.1 BFS

Our breadth-first-search algorithm is implemented as a priority queue of states. It starts with only the initial state. Every time it visits a state, it explores other states that may be attained by one action, and adds them to the queue. The priority of the queue is set on the total travel costs to attain that state. If we explore a state that was previously explored, but with a lower cost, the value is updated. To detect cycles, a list keeps all states that have been seen.

It will raise an error if no goal state is found, though this may only happen if the initial input was not appropriate.

To explore states, we look at the available actions. These are: delivering a task that is being carried, or picking up a task that is not started. A task is not picked up if the vehicle does not have the capacity to carry it.

2.2 A*

The A* algorithm is very similar to the BFS except that the cost assigned to a state does not only depend on the travelling costs, but also on a heuristic function estimating the future costs.

2.3 Heuristic Functions

The heuristic functions try to estimate how much cost is required from each state to attain a final state. To guarantee an optimal solution, the function should never overestimate the remaining cost.

We start from the observation that every action (or state-change) will be a route travelled. From the tasks remaining there is a list of cities that must be travelled to.

-"current-distance": This heuristic only takes the considered state into account, and the very next. It will simply look at the shortest path to a city with a task.

-"all-distances": This heuristic takes every city that must be attained, and computes the cost related to travelling to a closest other city that must also be attained. For the current state, we compute as for "current-distance".

The above heuristics acts optimally but tend to underestimate too much the remaining cost.

-"greedy-path": This heuristic computes a path from the current state to a final state, without capacity limit: it will iteratively travel to the closest city with a task, until all cities with a task have been travelled through. This heuristic does not guarantee to be optimal. However in practice, we only obtained optimal solutions.

3 Results

3.1 Experiment 1: BFS and A* Comparison

We test different algorithm: BFS and ASTAR with its 3 different heuristics as described above.

We measure the performance of an algorithm by the number of states explored. We chose this over computation time measures as it is not hardware specific. We also show the total cost of the path to verify optimality.

3.1.1 Setting

The topology used is Switzerland. The number of task is fixed at 8. The weight of a task is fixed at 3. We test for different capacity of the vehicle: 3,6,9 and 12.

3.1.2 Observations

capacity / algorithm	BFS	A* 'current-distance'	A* 'all-distances'	A* 'greedy-path'
3	2'182 (15'050)	2'165 (15'050)	2'167 (15'050)	2'176 (15'050)
6	9'718 (9'750)	9'575 (9'750)	8'818 (9'750)	8'187 (9'750)
9	20'682 (8'750)	20'383 (8'750)	17'993 (8'750)	16'237 (8'750)
12	30'751 (8'550)	30'042 (8'550)	26'764 (8'550)	24'295 (8'550)

Table 1: Number of states explored (travel costs)

As expected we observe that the A* algorithm is more efficient. By using the more accurate heuristic: 'all-distances', the results are better than with the simple 'current-distance' heuristic. Same goes for the 'greedy-path' heuristic versus the two other. We also observe that the 'greedy-path' algorithm always yields the optimal solution though it does not guarantee it theoretically. This may be an advocate for the use of an algorithm that does not guarantee optimality, in the case that computation time is key.

3.2 Experiment 2: Multi-agent Experiments

We test the effect of having multiple agents for the same amount of tasks to perform. In this set-up the agents do not cooperate and might have to make a new plan when they notice the task they wanted to pick up is gone.

We measure the efficiency as the time taken to deliver all tasks.

3.2.1 Setting

The topology used is Switzerland. The number of task is fixed at 7. The weight of a task is fixed at 3. Every vehicle has the same capacity. We test for different capacity of a vehicle: 3,6,9 and 12.

The algorithm used for this experiment is not relevant as long as it is optimal.

3.2.2 Observations

capacity / nb of agents	1	2	3	4
6	3.25	1.8	1.75	1.65
12	3	2.25	2	2.2
24	3	1.8	1.8	1.8

Table 2: Time to deliver all task, x10^3 s.

First, we observe that adding one agent makes the system more efficient (in terms of time), and after that it saturates. The agents will just be 'racing' to pick up the same tasks. Second, we observe that increasing the capacity won't help. It may even have the opposite effect as a single agent will stack up tasks, leaving the other agents with no work.

If we add up the travel costs of each agent, we see that the more agents we deploy, the less cost efficient the system is.