Excercise 3
Implementing a deliberative Agent

Group №23: Jean-Thomas FURRER, Emily HENTGEN

October 21, 2017

1 Model Description

1.1 Intermediate States

In our model, a state is characterized by the following six attributes:

- int[] **tasksStatus**: keeps track of whether each task has not been picked up yet, has been picked up but not delivered, or has been delivered
- int **taskIndex**: the index of the task being currently handled in this state
- City **departure**: the current city where the agent's vehicle is
- double **cost**: the accumulated cost in this state
- double **charge**: the accumulated charge in this state
- State **previous**: the previous state the agent's vehicle was in

1.2 Goal State

A goal (or final) state is a state where all tasks have been delivered. For a given state, this can be checked using the *tasksStatus* attribute.

1.3 Actions

There are three possible actions in our model: moving from one city to a neighbouring city, picking up a task and delivering a task.

2 Implementation

2.1 BFS

The main issue with the BFS algorithm is that the number of states grows exponentially with the number of tasks. The implementation must hence create states occupying few memory and each state visit must be as fast as possible. To this end, we use a *LinkedList* for storing the the set of all states that are yet to be visited and *HashSet*s for storing the successors of the current state as well as the set of all the already visited states. In the previous *State* definition, we further use an array of *int* for keeping track of each task status: it occupies few memory, and updating it induces less overhead than *List*s or *Set*s for instance storing the yet to be picked up/picked up but not yet delivered tasks. Following the same idea, instead of storing the complete list of actions that lead to the state, we store a reference to the previous state, which is less memory consuming.

2.2 A*

The A* algorithm is similar to the BFS algorithm. The main difference is that the queue of states is reordered as soon as new (successors) states are added to it. Moreover, once a final state is reached, the algorithm does not further visit other states, but terminates and uses this final state for building a plan.

Algorithm 1 BFS

```
INPUT
    Vehicle $vehicle$                                   ▷ the agent's vehicle
    TaskSet $tasks$                                     ▷ the set of tasks to deliver
OUTPUT
    Plan $plan$                                         ▷ the plan found by the BFS algorithm

$Q \leftarrow \{initialState\}$, $S \leftarrow \{\}$, $loopCheck \leftarrow \{\}$
$minimumCost \leftarrow \infty$, $finalState \leftarrow null$
while $Q$ is not empty do
    $currentState \leftarrow Q.pop$
    if $currentStat$ is a final state then
        if $currentState.cost < minimumCost$ then
            $minimumCost \leftarrow currentState.cost$
            $finalState \leftarrow currentState$
        end if
    end if
    if $loopCheck$ does not contain $currentState$ then
        $loopCheck \leftarrow loopCheck \cup \{currentState\}$
        $S \leftarrow currentState.successors$
        $Q \leftarrow Q \cup S$
    end if
end while
$plan \leftarrow buildPlan(finalState, plan, tasks)$
return $plan$
```

2.3 Heuristic Function

3 Results

3.1 Experiment 1: BFS and A* Comparison

The BFS algorithm runs out of memory when the number of tasks to deliver exceeds 8; the A* algorithm can go up to 9 tasks before it throws an *OutOMemory* exception as well. More detailed performance results for both algorithms 6 up to 9 tasks are shown in the Table *.

Number of tasks		6	7	8	9	10		6	7	8	9	10
Minimum cost		?	?	?	-	-		?	?	?	?	-
Number of states	BFS	?	?	?	-	-	A*	?	?	?	?	-
Execution time (ms)		?	?	?	-	-		?	?	?	?	-

In overall, since it visits less states, the A* algorithm takes less time to compute an optimal plan the BFS algorithm finds by visiting all the possible states.

3.1.1 Setting

3.1.2 Observations

3.2 Experiment 2: Multi-agent Experiments

3.2.1 Setting

3.2.2 Observations