Excercise 3
Implementing a deliberative Agent

Group №23: Jean-Thomas FURRER, Emily HENTGEN

October 24, 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
- 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

Each state additionally keeps a reference to the List<Task> *tasks* of the tasks that have to be delivered, an int *taskIndex* indicating which task is being currently handled in this state, as well as a *State* reference to the previous state the agent's vehicle was in: this is used for backtracking the list of actions the agent has to perform in order to reach this state, in particular when this is a final state.

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 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

```
1: Q ← {initialState}, S ← {}, loopCheck ← {}
2: minimumCost ← ∞, finalState ← null
3: while Q is not empty do
4:     currentState ← Q.pop
5:     if currentStat is a final state then
6:         if currentState.cost < minimumCost then
7:             minimumCost ← currentState.cost
8:             finalState ← currentState
9:         end if
10:    end if
11:    if loopCheck does not contain currentState then
12:        loopCheck ← loopCheck ∪ {currentState}
13:        S ← currentState.successors
14:        Q ← Q ∪ S
15:    end if
16: end while
    plan ← buildPlan(finalState, plan, tasks)
17: return plan
```

2.3 Heuristic Function

The main idea behind our heuristic function is that we want to minimize the final cost. In the BFS algorithm, once we reached all the final states, we keep only one which is the one having the smallest cost. Hence the simplest heuristic function can simply be to compare the cost of reaching two states. In that way, we can be sure that when reaching the very first final state, this is also the optimal one and we can stop the algorithm.

3 Results

3.1 Experiment 1: BFS and A* Comparison

3.1.1 Setting

We use at first the default configuration with uniformly distributed tasks, the map of Switzerland, a unique agent starting from Lausanne, and a seed of 23456. Afterwards, we also use a seed of 65432.

3.1.2 Observations

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 (in less than a minute) before it throws an *OutOMemoryError* as well, or times out. More detailed performance results for both algorithms with 6 up to 10 tasks and a seed of 23456 are shown in the Table 1.

Number of tasks		6	7	8	9		6	7	8	9	10
Minimum cost		6900	8050	8550	-		6900	8050	8550	8600	-
Execution time (ms)	BFS	113	1155	9723	-	A*	102	230	2146	18147	-
Number of states		49 993	435 391	2 517 570	-		22 097	222 660	1 351 656	6 420 867	-

Table 1: Performance of BFS and A* for a seed equal to 23456

By changing the seed to 65432 for instance, the number of tasks for which the agent can compute a plan increases:

the BFS agent can now handle up to 9 tasks, and the A* agent up to 10. We notice the cost for 9 and 10 tasks is the same with the A* algorithm: as the number of tasks increases, the probability that the agent will have to pick up or deliver a task to a city it previously only passed through increases as well. If its capacity allows it, it can thus design a plan which includes the additional task with a cost no higher than before (Table 2).

Number of tasks		7	8	9	10		7	8	9	10	11
Minimum cost		5700	6600	7150	-		5700	6600	7150	7150	-
Execution time (ms)	BFS	260	3122	31 627	-	A*	68	472	4501	52 153	-
Number of states		115 157	991 288	5 307 498	-		46 464	437 334	2 488 459	10 144 872	-

Table 2: Performance of BFS and A* for a seed equal to 65432

For this seed, BFS is able to build a plan in less than one minute for up to 9 tasks, and A* for up to 10 tasks. This number can vary depending on the initial setting and the seed though. It can also slightly change across different simulations, although the initial configuration is the same.

Compared to the BFS algorithm, the A* algorithm can only design a plan with one additional task before tt eventually times out or throws an *OutOfMemoryError*. However, in overall, since it visits less states, the A* algorithm also takes less time to compute an optimal plan the BFS algorithm finds by visiting all the possible states.

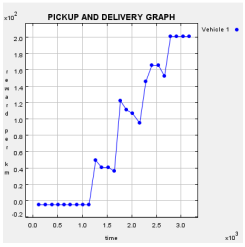
3.2 Experiment 2: Multi-agent Experiments

3.2.1 Setting

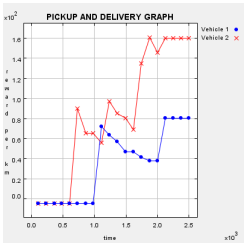
We use the default configuration, with the map of Switzerland, and a seed of 23456. In this setting, there are up to three agents competing: the first agent starts from Lausanne, the second from Zürich and the third from Bern.

3.2.2 Observations

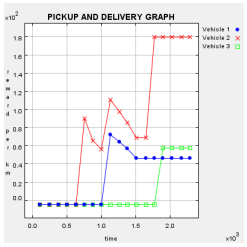
As the number of agents increases, their average reward per km decreases, since they are not coordinated. Moreover, one plan recomputation induces a cascade of other plan recomputations: after the agent starting from Zürich performs in the beginning one plan recomputation, the agent starting from Lausanne ends up doing four plan recomputations. Each new plan involves a task the agent from Zürich happens to just have picked up, so in the end, the agent from Lausanne delivers only the task it picked up before the conflict occurred. With fewer tasks, four for instance, one agent may end up not delivering any tasks, as shown in Figure 1 (d).



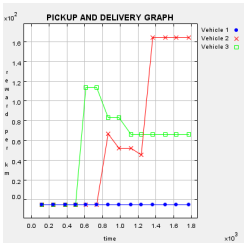
(a) One deliberative agent using the BFS algorithm, 7 tasks, seed 23456



(b) Two deliverative agents using the BFS algorithm, 7 tasks, seed 23456



(c) Three deliberative agents using the BFS algorithm, 7 tasks, seed 23456



(d) Three deliberative agents using the BFS algorithm, 4 tasks, seed 23456

Figure 1: Competing deliberative agents using the BFS algorithm

When there are multiple agents which do not coordinate their actions, they end up displaying a worse performance (in terms of reward per km) compared to the same setting where each of them would operate alone.

Theoretically, an agent using the A* algorithm should outperform an agent using the BFS algorithm, since it could begin picking up tasks sooner, while the other agent would still be computing its optimal plan. The platform however does not allow us to observe this phenomenon, because the agents start moving when all agents are done computing their plan.