Excercise 3
Implementing a deliberative Agent

Group 23 : Alessandro Bianchi, Davide Nanni

October 20, 2020

1 Model Description

1.1 Intermediate States

`State` ::= ⟨ `currentCity` , `residualCapacity` , `pickedUpTasks` , `availableTasks` ⟩
where `currentCity` is the `City` in which the agent is for the state, `residualCapacity` is the capacity of the vehicle available for new tasks, `pickedUpTasks` is the `Set` of `Task` s the agent has picked up and still has to deliver, and `availableTasks` is the `Set` of `Task` s available for pick up.

1.2 Goal State

A state is a goal state if all tasks have been delivered, i.e. both `pickedUpTasks` and `availableTasks` are empty.

1.3 Actions

An action is identified by the following properties:

- `City destination` : destination of the agent's movement. This is not necessarily a neighbor to `currentCity` , as this is a high level abstraction.
- `Set<Task> pickedUpTasks` : tasks the agent picks up at `currentCity` . Can be empty if no tasks were picked up. The sum of the weights of the picked up tasks must be less than `residualCapacity` .

The transitition is defined as transition(s, a) :

- `newCurrentCity` : `a.destination`
- `newPickedUpTasks` : (`s.pickedUpTasks` ∪ `a.pickedUpTasks`) \ set of tasks with delivery city = `newCurrentCity`
- `newResidualCapacity` : *vehicle capacity* − sum of weights of tasks in `newPickedUpTasks`
- `newAvailableTasks` : `s.availableTasks` \ `a.pickedUpTasks`

2 Implementation

Three main functions are shared between **BFS** and **A***:

- `getInitialState` returns the initial `State` of an execution. If `planCancelled` was called previously, it uses the information about the carried tasks to initialize the state.

- `computeDerivedStates` returns, given a state s, the set of "children" states reachable from s with a valid AND useful action (that is, any action a from s to any city $\neq$ `s.currentCity` to which the agent has at least one task to deliver or which has at least one task ready for pick up and with `a.pickedUpTasks` = the maximum subset of tasks the agent can pick up from `currentCity` considering its `residualCapacity`).

- `fillPlan` is a recursive method, which computes and returns an optimal plan given an optimal final state. In order to perform the recursive steps, the method uses `State` 's additional attribute `previousChainLink` , which contains the previous `State` and the `DeliberativeAction` executed to move from that to itself. This way, all the states up until `initialState` are reached. At this point, the actions are executed in order and the state transitions applied, up until `finalState` is reached again.

2.1 BFS

The classic algorithm was implemented, with the search stopping at the best final state found on the first layer with final states. In order to shorten the search time, we keep a `Map<State, Double>` of visited states together with their best cost. When a new state is to be added to the queue, we only add it if it is not already in the map or if it is already present, but the newly found state has a better cost. We then update the cost in the map. This way, we prune branches that would lead to non-optimal executions.

2.2 A*

The classic algorithm was implemented. The queue is implemented as a `SortedQueue` initialized with a comparator on sum(`state.chainCost` , `heuristic(state)`) for the same state.

2.3 Heuristic Functions

We devised two heuristic functions:

- **H1**: distance from `currentCity` to the farthest `destination` of any task

- **H2**: maximum between the maximum distance to deliver a picked up task and the maximum distance to pick up and deliver an available task.

We will hereby prove the monotonicity/consistency property for the heuristics:

- **H1**: we want to prove that, for any x,y, $h(x) \leq d(x,y) + h(y)$. In order to do this, we try to minimize the right-hand expression. In the worst case scenario, the movement (of length d) will minimize $h(y)$. As a consequence, the distance between the current city of the state y and the farthest destination city (which is not necessarily the same as in the previous state) will at most decrease by d. In such a case, $h(x) = d + h(y)$. Otherwise, the movement will either decrease the max distance by a value $< d$, keep it the same or increase it, thus $h(z) < d + h(x)$ $\square$

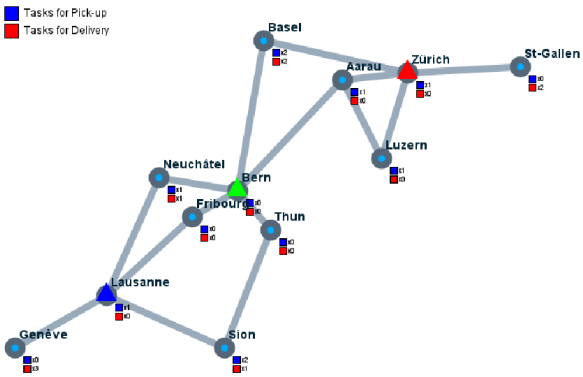
- **H2**: a similar proof can be built for **H2**. In this case, however, the measure that cannot decrease by a factor $> d$ is the total distance to be traveled to deliver the most expensive task, considering both delivery cost and, where it applies, pickup cost.

Both heuristics should therefore lead to an optimal solution.

3 Results

Both the experiments were run on similar conditions: the Figure shows the chosen topology, the distribution of tasks (for 9 tasks, Experiment 2) in the topology and the home cities of each agent. The blue position is used when only one agent is running.

		Task number			
		6	8	10	11
Iterations	BFS	3047	66742	784869	-
	A* H1	1441	35504	434081	-
	A* H2	593	16892	196137	480469
Planning time (ms)	BFS	230	1658	18740	timeout
	A* H1	221	2195	29426	timeout
	A* H2	113	929	9919	28080
Minimal cost	BFS	6900	8550	9100	-
	A* H1	6900	8550	9100	-
	A* H2	6900	8550	9100	9100



3.1 Experiment 1: BFS and A* Comparison

3.1.1 Setting

In this experiment, a single agent was run with the different algorithm/heuristic combinations and with different numbers of tasks, in order to compare their optimality and performance.

3.1.2 Observations

As we can see in the Table, both the BFS and A* come to the same optimal result, although the former is slower than the latter with **H2**. **H1** still takes less iterations than BFS to compute, but is slower nonetheless. **H2** speed allows to find the optimum under 60 seconds for 11 tasks, compared to the 10 tasks of BFS and **H1**. Anyway, this does not influence the agents' efficiency in a competitive scenario, as the whole simulation pauses when any agent recomputes its plan.

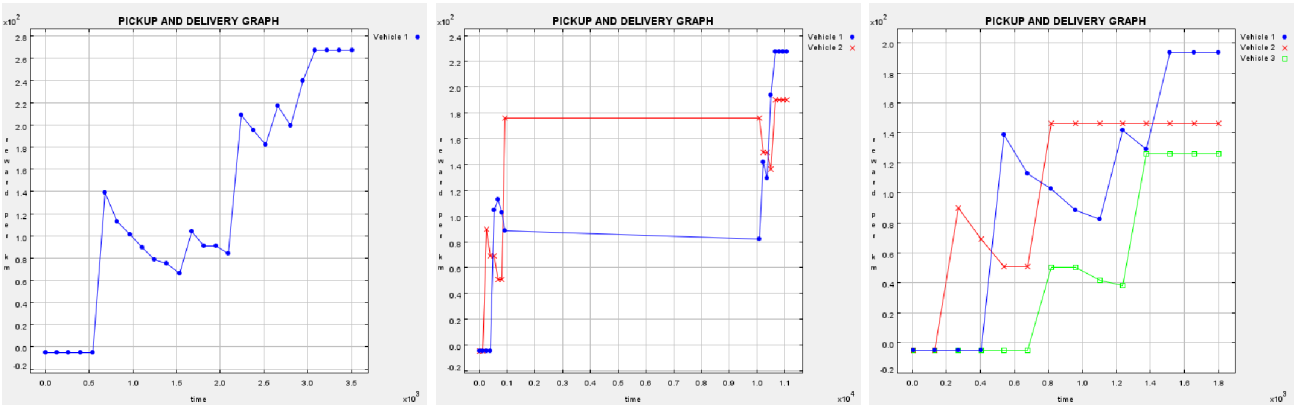
3.2 Experiment 2: Multi-agent Experiments

3.2.1 Setting

In this experiment, one, two, and finally three agents were run concurrently with 9 tasks available.

3.2.2 Observations

As we can notice from Figure 1b and Figure 1c, the presence of multiple concurrent agents leads to a decrease in performance for each one, as the number of available tasks is limited. However, this decrease is less than harmonic in the number of agents, as their ability to recompute the plan prevents them to perform useless actions after the plan is interrupted.



(a) One agent (b) Two agents (c) Three agents