Excercise 4
Implementing a centralized agent

Group 47 : Elia Anzuoni (312827), Francesco Ballestrazzi (312494)

November 3, 2020

1 Solution Representation

1.1 Variables

We chose to represent a solution via a single associative array (implemented as a `Map`), called `firstActions`, mapping each vehicle to the `Node` containing its first action.
The `Node` class is similar to the one used for a doubly-linked list: it holds an element (an action, in our case), a link to its predecessor, and one to its successor.
The action was implemented as a class (called `Azione`), holding the task concerned by the action, and the type of the action (either `PICKUP` or `DELIVERY`).
In summary, a solution is uniquely identified by its `firstActions` associative array, which implicitly describes, for each vehicle, its sequence of "non-trivial" actions (i.e. only pickups and deliveries): the `Moves` are derived from these, as every vehicle is always made to move on the best path between two cities.

1.2 Constraints

For a solution to be admissible i.e. for it to lead to an admissible joint plan, the following conditions need to be met:

- For every task t, there must exist exactly one pickup action p and one delivery action d in the whole solution (i.e. in the union of the vehicle's action sequences) having t as task; moreover, p and d must both appear in exactly one vehicle's action sequence - the same one -, and p must appear before d.
- For any vehicle v and any action a in v's action sequence, it must hold that $c(v) \geq w(v, a)$, where $c(v)$ is the capacity of v, and $w(v, a)$ is the weight carried by v after it performs action a (i.e. the sum of the task weights over all pickups up to a minus the sum of the task weights over all deliveries up to a).

The first constraint, called integrity, ensures that the solution corresponds to a joint plan which "makes sense" (where all tasks are taken care of, no two vehicles take care of the same task and so on). The second constraint, called admissibility, ensures that each vehicle's action sequence is compatible with the vehicle's capacity.
These constraints are never explicitly checked. Instead, the initial solution is constructed so as to respect them, and the neighbours are generated by functions that preserve them.

1.3 Objective function

The objective function, to be minimised, is the cost of the joint plan, i.e. the sum of the costs of each vehicle's action sequence. The cost of an action sequence is the cost per km of the vehicle, multiplied by the sum, over pairs of consecutive actions in the sequence, of the distances (along the best path) between the cities where the actions take place (an action takes place in its task's pickup city, if it is a pickup, otherwise in its task's delivery city); the first city is taken to be the home city of the vehicle.

2 Stochastic optimization

2.1 Initial solution

The initial solution is, at first, generated deterministically: tasks are assigned trivially (i.e. with the delivery placed immediately after the pickup) to the first vehicle whose capacity is not exceeded by the task's weight. The vehicles

are not iterated over always in the same order, to prevent the first ones to be assigned many more tasks: instead, the iteration resumes from the vehicle after the one where the previous iteration stopped (circular iteration).
After this first phase, a dequence of calls is made to `getRandomNeighbour`, first, and to `getBestNeighbour`, after.

2.2 Generating neighbours

Two (optimised) methods are available to explore the surroundings of a solution: `getRandomNeighbour` and `getBestNeighbour`. They return, respectively, a random and the best among a (random) set of neighbours, without saving it all in memory and then choosing. The random set of neighbours is defined as follows.
A task t and a vehicle u are chosen at random, such that $u \neq v$ (where v is the vehicle currently taking care of t), and $c(u) \geq w(t)$ (where $w(t)$ is the weight of t); if this is not possible, then u is set to v. The random set of neighbours is then defined to be the union, over $w \in \{u, v\}$, of the sets of all admissible reassignments of t from v to w. Ssuch a reassignment is a solution identical to the current one, except that t is taken care of by w, and the order of pre-existing actions of w is preserved by the insertion of t's pickup and delivery.

2.3 Stochastic optimization algorithm

Our algorithm is an ”ϵ-greedy with reset” modified version of the SLS. This means that, at each iteration, we move to a random neighbour with probability ϵ, and to the best neighbour with probability $1 - \epsilon$. Moreover, the best solution seen so far is kept at all times: if more than `ITERSRESET` iterations pass by without an improvement of the best solution, the current solution is reset to the best solution, to ”restart” the search from there.

3 Results

3.1 Experiment 1: Model parameters

We present here the results obtained by changing the values of 2 parameters of our algorithm: ϵ and `ITERSRESET`. Both parameters are useful to control the level of exploration of the solutions space.

3.1.1 Setting

$\epsilon = \{0.2, 0.5\}$; $\texttt{ITERSRESET} = \{300, 1500, 5000\}$; $timeout = \{30s, 200s\}$; $Topology = England$; $\#tasks = 30$; $\#vehicles = 4$; $Taskconfiguration = default$

3.1.2 Observations

By setting a small `ITERSRESET` we achieve good results fast. However, doing so may not give enough time to properly explore the space before resetting to the previous best solution. This is fine when the timeout is small, because we want to quickly abandon not-so-promising solutions, but may result in worse results in the long run.

Overall we obtain the best cost with $\epsilon = 0.2$ and $itersReset = 300$, when running on 30 seconds, and with $\epsilon = 0.5$ and $itersReset = 1500$ on 200 seconds.

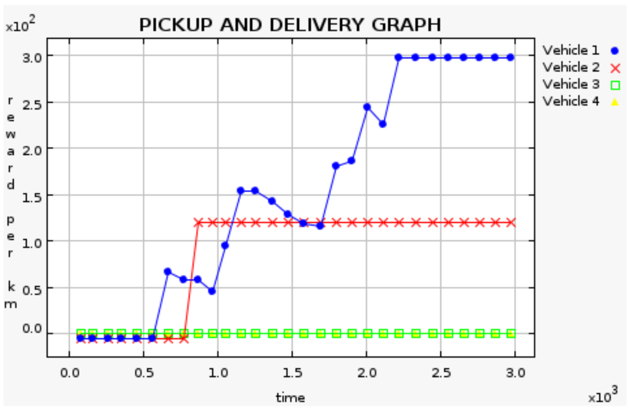
ϵ	*itersReset*	*timeout*	*planCost*
0.2	300	30s	21936
0.2	1500	30s	35101
0.2	5000	30s	44108
0.5	300	30s	23253
0.5	1500	30s	24412
0.5	5000	30s	47672
0.2	300	200s	21936
0.2	1500	200s	23513
0.2	5000	200s	33787
0.5	300	200s	22029
0.5	1500	200s	20814
0.5	5000	200s	24939

3.2 Experiment 2: Fairness of the plan

3.2.1 Setting

$\#tasks = \{10, 30, 60\}; \#vehicles = 4; \epsilon = 0.2; iterReset = 300; timeout = 30s; Topology = England; Taskconfiguration = default$

3.2.2 Observations



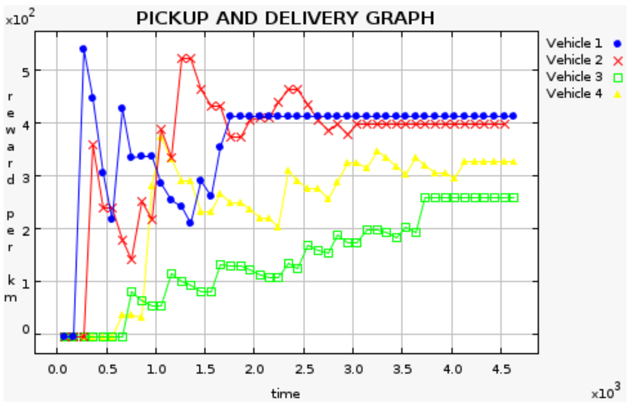


Figure 1: On the left the reward/km with 10 tasks. On the right, with 60 tasks.

We can see that the number of tasks plays a major role in influencing the load balancing among the vehicles. When we have few tasks, some of the vehicles won't move at all, while when the number of tasks increases, every vehicles gets more or less the same work to do. Thus, optimality and fairness do not always coincide.

3.3 Experiment 3: Cost with different #vehicles

We analyse the effect of the number of available vehicles.

3.3.1 Setting

$\epsilon = 0.2; iterReset = 300; timeout = 30s; Topology = England; \#tasks = 30; \#vehicles = \{2, 6, 12\}; Taskconfiguration = default$

3.3.2 Observations

$\#vehicles$	$planCost$
2	25189
6	25329
12	23126

The costs stays roughly constant, even when we add many vehicles. In fact, many of the additional vehicles are not used at all, causing a big difference in workload.

3.4 Asymptotic complexity

The complexity of the algorithm is completely specified by the complexity of a single iteration.
The complexity of `getRandomNeighbour` is $O(V + T + T^2) = O(V + T^2)$, since $O(V + T)$ is the cost of choosing a random task t and a random vehicle u; $O(T^2)$ is the number of reassignments of t to w: we generate each reassignment in constant time, because of the constant-time node operations we implemented.
The complexity of `getBestNeighbour` is $O(V + T + (V + T)T^2) = O((V + T)T^2)$; unlike before, every time we generate a reassignment, we check whether it beats the best solution and, if it does, we copy it into the best solution: this takes $O(V + T)$ time. Before, we only checked whether we had generated as many reassignments as indicated by a previously-sampled random counter, to decide whether to directly return the current one or to discard it once and for all.