Excercise 4
Implementing a centralized agent

Group 42 : Niederhauser Thibault, Stefanini Niccolo

November 3, 2020

1 Solution Representation

1.1 Variables

Importantly, for the variables definition, each existing task t is split into two objects t_{pickup} and $t_{deliver}$, which correspond to the pick-up and delivery of the task. Those are implemented in a new class, *PUDtask*, which contains a task and a type (deliver or pick-up).
The constraint satisfaction problem is modelled using 4 different variables:

- $nextTaskT = [nextTaskT(t_{1,pickup}), ..., nextTaskT(t_{N_T,pickup}), nextTaskT(t_{N_T,deliver})]$
- $nextTaskV = [nextTaskV(v_1), ..., nextTaskV(v_{N_V}))]$
- $time = [time(t_{1,pickup}), time(t_{1,deliver}), ..., time(t_{N_T,pickup}), time(t_{N_T,deliver})]$
- $vehicle = [vehicle(t_{1,pickup}), vehicle(t_{1,deliver}), ..., vehicle(t_{N_T,pickup}), vehicle(t_{N_T,deliver})]$

nextTaskT is an array of size $2N_T$ (N_T = number of tasks). It contains one instance for each PUDtask, i.e. two variables per task. Each element of *nextTaskT* corresponds to the task that will be taken care of by the same vehicle after the current task. It takes an existing PUD task or NULL as a value.
nextTaskV is an array of size N_V (N_V = number of vehicles). For each vehicle it contains one element: the first task that the vehicle deals with. It takes an existing PUD task or NULL as a value.
time is an array of size $2N_T$ and contains one integer for each existing PUDtask. Each element corresponds to the order of execution of the task.
vehicle contains one element for each PUDtask. The value of a variable is the vehicle that deals with the corresponding PUD task.

1.2 Constraints

Let t be a PUD task, i.e a task with a type: either "deliver" or "pick-up".

1. $nextTaskT(t) \neq$: the PUD task adressed after some PUD task t cannot be the same PUD task..
2. $nextTaskV(v_k) = t_j \Rightarrow time(t_j) = 1$: the tasks in *nextTaskV* are the first tasks to be dealt with.
3. $nextTaskT(t_i) = t_j \Rightarrow time(t_j) = time(t_i) + 1$: the *time* array is consistent with the *nextTaskT* array.
4. $nextTaskV(v_k) = t_j \Rightarrow vehicle(t_j) = v_k$: the *vehicle* array is consistent with the *nextTaskV* array.
5. $nextTask(t_i) = t_j \Rightarrow vehicle(t_j) = vehicle(t_i)$: the *vehicle* array is consistent with the *nextTask* array.

6. All tasks must be delivered: the set of values of the variables in the $nextTaskT$ and $NextTaskV$ arrays must be equal to the set of PUDtasks T plus N_V times the value NULL.

7. The capacity of a vehicle cannot be exceeded: if $type(t_i) =$ "$pickup$" and $load(ti) > remainingCapacity(vk) \Rightarrow vehicle(t_i) \neq v_k$

8. $time(t_{pickup}) < time(t_{deliver})$: A vehicle cannot deliver a task that has not been picked-up yet.

9. $vehicle(t_{pickup}) = v_k \Leftrightarrow vehicle(t_{deliver}) = v_k$: A task has to be picked-up and delivered by the same vehicle.

1.3 Objective function

The objective function is the total cost of the company: $C = \sum_{i=1}^{N_V}(totalDist(v_i) \cdot costPerKm(v_i))$

2 Stochastic optimization

2.1 Initial solution

For each existing task t, both corresponding PUD tasks t_{pickup} and $t_{deliver}$ are assigned to the "capable" vehicle (i.e vehicle's capacity > task's weight) whose home city is closest to the task's pick-up city. t_{pickup} and $t_{deliver}$ are always added consecutively and at the beginning of the selected vehicle. Note that if for any task, no big enough vehicle exists, the problem is unsolvable.
This initial solution is based on the assumptions that a good solution would be that each vehicle takes care of a cluster of nearby existing tasks and that distributing the tasks over several vehicles reduces the total cost.

2.2 Generating neighbours

First an "active" vehicle (i.e. a vehicle that takes care of at least one task) v_i is randomly selected and two local operators are applied to it: **1.** *Changing vehicle*: the first PUD task of v_i is selected. Both corresponding t_{pickup} and $t_{deliver}$ actions are removed from v_i and added consecutively at the beginning of another vehicle v_j. One neighbour is generated for each existing vehicle $v_j \neq v_i$. **2.** *Changing tasks order*: two tasks of v_i are exchanged respecting constraints. One neighbour is generated for each possible permutation.

2.3 Stochastic optimization algorithm

The stochastic optimization algorithm is based on the SLS algorithm described in the course. First the initial solution is computed. Then neighbours are generated and one of them is selected. The neighbours of the selected one are then computed again looping until a stopping criteria is reached.
The *LocalChoice* function is used to **select a neighbour** as a variable's assignment. The function selects the neighbour that minimizes the objective function and then returns it with a probability p. With a probability $1 - p$, another neighbour, chosen randomly, is returned. This allows to explore the objective function and increases the robustness against local minimum trapping.
The **stopping criteria** for the loop mentioned above is the following: if the best cost has not changed for n iterations, the loop is exited and the variables' assignment corresponding to the best cost during the whole optimization is returned as a solution.
Finally, due to the exploration probability $1 - p$, the chosen solution might be near a local minimum but not exactly at the local minimum. To slightly improve the final solution, the SLS algorithm is run one more time on the solution with $p = 1$ and $n = 1$. This last computation is rapid and ensures that the returned solution reaches the local minimum.

3 Results

3.1 Experiment 1: Model parameters

We want to explore the trade off between termination criteria (called iter) and exploration probability p, where with p=1 the agent always the best neighbors and with p=0, the agent always chose random. Easier stop criteria would give the agent more time to explore, but make simulation longer; lower exploration will make the agent converge faster to a local minimum.

3.1.1 Setting

Parameters: P = [0.2,0.4,0.6,0.8,1], iter=[100,1000,10000,50000]
Observables: computation **time** required for the simulation to converge (in seconds), final **cost** for the agent to complete the plan (in thousands).

3.1.2 Observations

P	0.2	0.4	0.6	0.8	1	0.2	0.4	0.6	0.8	1
Iter	100					1000				
Cost	31.2	31.1	29.2	33.7	33.7	30.4	28.1	29.4	26.6	34.8
Time	1.1	1.1	1.0	0.2	0.3	4.8	2.6	2.3	3.2	1.2
Iter	10000					50000				
Cost	29.4	28.4	27.8	27.5	34.8	28.3	28.1	26.3	26.3	34.8
Time	16.8	20.5	7.4	7.5	4.3	109	42.3	41.5	51	14.4

As expected, p=1 gives equally bad results for all the settings and p=0 (not reported in the table) never converges to a solution in time. When having enough iterations to explore (iter ¿ 1000), the results are similar but the time required to converge change heavily for small improvements of performance. It seems that p=0.8 gives the best results generally being a good trade between exploitation and exploration.

3.2 Experiment 2: Different configurations

W

3.2.1 Setting

We use the following configuration: map=England, ***task weight = 10***, vehicle capacity = 30, **p = 0.5**, **n = 10000**.

3.2.2 Observations

nb Tasks	10				20				30			
nb Vehicles	1	2	3	4	1	2	3	4	1	2	3	4
Cost v_1		0	0	0		0	0	18772			0	25917
Cost v_2		9866	9291	7021		24660	17802	1585			35080	1585
Cost v_3	X	X	0	0	X	X	3992	2462	X	X	0	7554
Cost v_4	X	X	X	3626	X	X	X	575	X	X	X	0
Total Cost		9866	9291	10 646		24660	21795	23394			35080	35056
Time [s]		14	11	25		73	60	53			287	178