Centralized agents

Jeremy Gotteland & Quentin Praz

November 12, 2015

Encoding the problem

We create the class *ExtendedTask* which is the extension of a basic task provided by *Logist*. This new class is used to differentiate the pickup action and the delivery action. It is characterize by a task and by an action (pickup or delivery). We used these *ExtendedTask*s to be able to carry multiple tasks in the same vehicle.
We also implement the class *Solution* which represents a possible solution. A solution is characterized by a map between a vehicle and a list of *ExtendedTask*s that we called *tasksList*. We have therefore one *tasksList* per vehicle.
The notion of time is defined by the indices of the task in the *tasksList*.

Constraints

To have a valid solution, we need to satisfy constraints:

1. A specific *ExtendedTask* can be contained in only one *tasksList*.
2. Let take the two *ExtendedTask*s coming from the same task (pickup and delivery part), the index of the pickup part has to be strictly smaller that the index of the delivery part.
3. Let take again the two *ExtendedTask*s coming from the same task, the two *ExtendedTask* have to be in the same *tasksList*.
4. All task must be delivered. The sum of the *ExtendedTask*s from all *tasksList*s had to be equal to the number of initial tasks multiplied by 2.
5. The weights of the carried tasks by a vehicle cannot exceed the capacity of this vehicle.

Generating the plans

We use the **Stochastic Local Search** algorithm to determine the plan of each vehicle, leading to an optimal solution. This algorithm works iteratively: starting from an initial solution, each steps creates a new set of solutions based on the current chosen solution (neighbours) and selects the one that gives the best cost function.

Local neighbours generation

In order to perform the SLS algorithm, we need to generate a set of solutions that are similar to the one we are iterating on.

To generate neighbours, we first select a random vehicle which has at least one task to pickup/deliver. Then we apply to it the two following **operators**

- Swap task **between** vehicles:
  For every other vehicle, generate solution by replacing the two first actions of each vehicle by the pickup-delivery of the **first pickup of the other one**
- Swap tasks **within** a vehicle:
  Generate as many solutions as there are possible combinations to swap **task order** within the random vehicle we chose.

From those we select the ones that are valid, and then compute their cost using the cost function described in the exercise sheet.
We select the solution that minimizes this objective function (if several solutions have the same value, we select one of them randomly).

Then comes into play a probability factor **p** that we set in the beginning **(here we chose the value 0.5)**. With probability **p**, we select the solution chosen before and continue SLS with this value as our current solution. But with probability **(1 - p)**, we stay in the same state and run again the algorithm, with the current state not being changed.

Note that we limit the number of iterations of SLS to **5000 iterations**.

The importance of the initial solution

Our initial solution was to give task 1 to vehicle 1, task 2 to vehicle 2, etc.. . If we have 4 vehicles, we give task 5 to vehicle 1 (given that he has enough capacity) and we continue until all tasks are assigned.

This proved to converge to the right solution relatively quickly, as well as having a low computation time because the number of combinations of tasks within

each vehicle was limited.

However, starting from an initial solution where all tasks were put in one vehicle gave much better result regarding to the cost, but had a much higher computation time.

Result of the comparison are displayed in the **Results** section.

Results

This graph contains all of our results. We ran the algorithm for 10, 20, 30 and 40 tasks, with everytime 1, 2, 3 or 4 vehicles.

In the case of the initial solution where all tasks are put in one vehicle, they always stay in that same vehicles so we do not need to differentiate.

However it is important in the case where the initial solution tries to spread tasks in every vehicle.

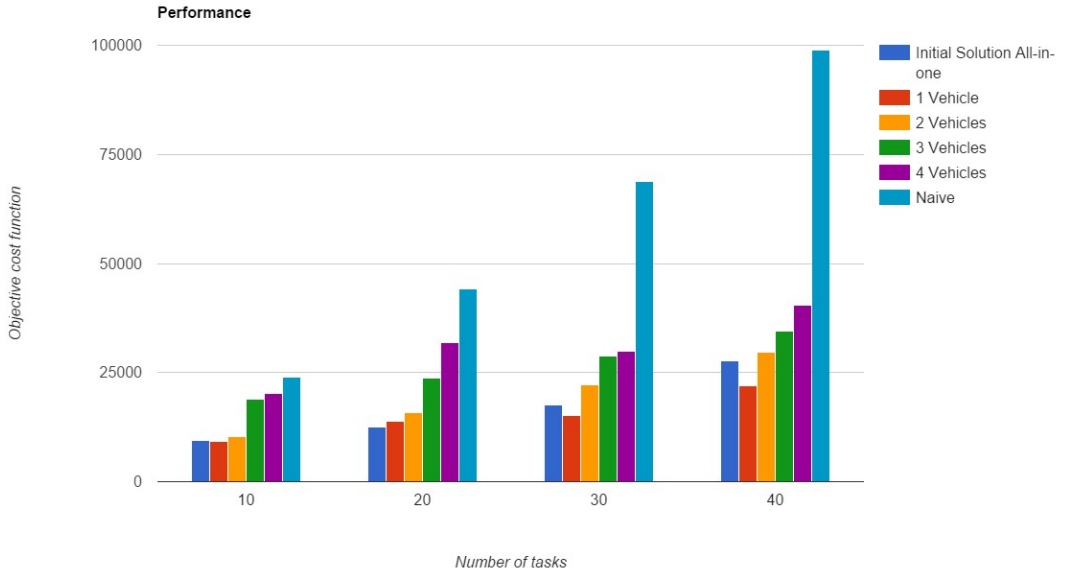

Figure 1: Performance in terms of cost for each combinations of parameters/algorithm

Number of tasks	All-in-one	1 Vehicle	2 Vehicles	3 Vehicles	4 Vehicles	Naive
10	9549	9196	10222	18890	20136	24029
20	12487	13855	15761	23732	31876.5	44214
30	17600	15241	22227	28739	29847	68731
40	27756	22027	29744	34488	40376.5	98896

Table 1: Values of the barplot