Excercise 4:
Centralized Coordination

Group №1 : Célia Benquet - 271518, Artur Jesslen - 270642

November 3, 2020

1 Solution Representation

1.1 Variables

We represented the solutions through a **Solution** class. One solution corresponds to a possible set of actions for all vehicles to perform in a possible order. Hence, the **Solution** class is defined through 5 public attributes. **nextTaskForVehicle** maps each vehicle to the first task it has to take in charge (pickup). Then, **nextTaskForTask** maps each task to the next one. It is implicit that following tasks are taken in charge by the same vehicle. The schedule of the actions sequence is set in **pickupTimes** and **deliveryTimes** through Integer time steps. Each task is mapped to a time step at which to pick up in **pickupTimes** and deliver in **deliveryTimes**. Finally, each task is also mapped to the vehicle that takes it in charge in **vehicles**.

An instance of a solution can generate its corresponding set of plans, one for each vehicle of the environment, based on its attributes, through **generatePlan**.

1.2 Constraints

To be valid, a solution must allocate values to its attributes in order to satisfy some constraints, defined in **isValid**. They are the following.

- The task delivered after some task t cannot be the same task t.
- The task delivered after some task t cannot have a smaller pickup time than the pickup time of t. It is however possible to deliver it before the delivery of t.
- Each vehicle must pick up its first task during the first time step.
- Each vehicle must deliver a task after it picks it up. In other words, the delivery time of the task cannot be smaller than its pick up time.
- Each vehicle can only perform one action (either pick up or deliver) at a time (for one time step). For example, if tasks t and t' are taken by a same vehicle, they cannot have coinciding times of action (pickup or delivery).
- The first task of a vehicle must be handled by this vehicle.
- The task delivered after some task t must be handled by the same vehicle than t.
- All tasks must be delivered. This is checked by checking the size of the HashMap attributes. For instance, for n tasks in the environment and k vehicles, **nextTaskToTask** must contain n entries with k of them that are null values (if all the vehicles are attributed to at least one task), corresponding to the last tasks to be handled for each vehicle.

- The capacity of a vehicle cannot be exceeded, meaning that a vehicle cannot take more tasks than what its capacity allows.

1.3 Objective function

Because all the tasks are eventually delivered by the company, the rewards of the tasks will all be obtained, meaning the company shouldn't take them into account for the optimization. Consequently, the objective function only consist in minimizing the cost of the solution, while satisfying the constraints. In other words, we compute the sum of the individual cost of each plan for each vehicle.

2 Stochastic optimization

2.1 Initial solution

We generate the initial solution in **selectInitialSolution** in 2 different optional ways. The *slave* way consists in putting all the tasks in the vehicle with the highest capacity. The *distributed* way consists in distributing the tasks to all the vehicles of the environment, equally. The other attributes are filled consequently. The times to pickup and deliver are set to be consecutive actions for the vehicle handling the task. It means that when a vehicle picks up a task it will necessarily deliver it at the next time step.

2.2 Generating neighbours

We generate neighbours solutions based on the last optimal solution in **generateNeighbours**, following the Stochastic Local Search Algorithm for COP (Constraint Optimization Problem) requirements. We randomly generate similar solutions by switching the vehicle that handles a given first task for all vehicles in **changingVehicle** and shuffling the order of the tasks in each vehicle in **changingTaskOrder**. The generated solutions are checked right away to verify that they fit the constraints.

2.3 Stochastic optimization algorithm

The Stochastic Optimization Algorithm takes place in **computePlans**. First of all, we generate an initial solution to set the optimal solution using **selectInitialSolution** and based on our different initializations (*slave* and *distributed*). Then, for a given number of iterations, we optimize the solution. Neighbour solutions are generated through **generateNeighbours**. Then, **localChoice** chooses which solution is the new optimal. The newly generated solutions are sorted by the cost of their global plan and the method returns either the last optimal solution or the newly found optimal solution (lower cost of the plan), depending on a probability to explore/exploit.

3 Results

3.1 Experiment 1: Model parameters

3.1.1 Setting

We compared the different implementations (*slave* and *distributed*) used to compute the initial solution. Then, we performed the rest of the observations using the *distributed* initialization. We observed the resulting plans for 10 iterations compared to 10.000 iterations with a probability of exploration/exploitation of 0.4. Similarly, we compared results for a probability to explore new solutions or exploit the last optimal solution found of 0.1, 0.4 and 0.9 with 500 iterations. Finally, we compared different ways of generating the neighbouring solutions, either by changing both the vehicle attributed to a given task and the order of the tasks for a given vehicle or just the attribution of the tasks.

3.1.2 Observations

The *slave* initialization provides a cheaper global plan. We see that the optimal solution for the *slave* initialization consists of having one vehicle doing all the deliveries, as it minimizes the global cost. However, it seems to be stuck in a local minimum where all the tasks are handled by the same vehicle, even after a high number of iterations. For *distributed*, the algorithm seems to converge to solutions with higher cost compared to *slave* but they share the tasks. If we were to add a notion of time that the company takes to handle the delivery of all the tasks, which makes sense when one considers the client satisfaction, this algorithm would perform better. That is why we decided to keep the *distributed* implementation of the initialization.

Increasing the number of iterations in the stochastic search algorithm increases the chances to find the optimal solution as the generated neighbouring solutions are more and more shuffled. It also helps to have more exploration, rather than only exploitation of the initial solution or a really close one, which is non-optimal.

For the probability to explore/exploit, we resonate that a high value promotes exploration while a small value exploitation. Indeed, a value of 0.9 made the solutions to always change. It is more probable that the optimal solution gets trapped in a local minimum, however the algorithm will converge fast. A value of 0.1 makes the algorithm less exploratory and consequently converging slower.

Implementing the neighbour solutions using both changing methods, we see that even with a higher cost per kilometer for the vehicle with the highest capacity, the algorithm seems to be stuck in a local minimum when using the *slave* initialization. Indeed, the vehicle with the highest capacity, which has all the tasks in the initial solution, systematically keeps the tasks for itself instead of giving them up to cheaper vehicles. When we test the solution generation without changing the tasks ordering, the optimal solution ends up having more distributed task between vehicles as it's the only why to take the cost lower.

3.2 Experiment 2: Different configurations

3.2.1 Setting

We observed the behavior of the environment for 30 and 60 tasks and for 4 and 7 vehicles. We use the *distributed* initialization as it seems more fair to us from the beginning, as explained in Part 3.1. The probability of exploration/exploitation was set at 0.4 and the number of iterations at 10.000.

3.2.2 Observations

Using the *distributed* way of initialization, the complexity of the algorithm increases greatly with mostly the number of tasks but also the number of vehicles. Indeed, increasing the number of tasks or vehicles means increasing the number of generated solutions in **generateNeighbours**. For one added task, **changeTasks** has n more swaps to compute per call and for one added vehicle, **changeVehicle** has 1 more swap to compute per call.

Based on our observations, even when there are more vehicles or more tasks, the optimal solution consists of making only a few vehicles handling most of the tasks, while the order of the tasks for each is optimized. Hence, it generally looks like the algorithm converges to an unfair plan, where the tasks are not equally distributed between the vehicles. It makes sense as the cost is computed as the sum of all the travelled distances. Hence, having less vehicles active, even if it makes them travel more individually, makes the global cost lower.