Excercise 4
Implementing a centralized agent

Group №16: Léopold Bouraux, Vincent Rinaldi

November 5, 2019

1 Solution Representation

1.1 Variables

In order to find the optimal delivery plan for a centralized agent, we use different variables than those recommended in the problem statement: for each vehicle, we keep in memory a list of *TaskAction* to be performed. Each *TaskAction* has a task and an action linked to it through a boolean variable *isPickup*. Our goal here is basically to find the optimal mapping between each vehicle and a linked list of *TaskAction*. This entity is represented in the class *VehiclePlan*. Hence our *VehiclePlan* structure replaces the two arrays present in the statement, *nextTask* and *time*. Calculations performed for the stochastic optimization are in the *CSP* class, where there is a list of all the vehicles present and a map is created between each vehicle and its *VehiclePlan*.

1.2 Constraints

The following constraints apply throughout the optimization process for each plan:

- If a task is already assigned to a vehicle, then it cannot be given to another one.
- If a *TaskAction* i and follows a *TaskAction* j in the *VehiclePlan*' list, then Action j is performed just after Action i.
- If actions list of a *VehiclePlan* is empty, then this vehicle has no more task to perform.
- When a task is added to a plan, two actions are created.
- If a vehicle has a pickup *Action*, then it must have the deliver *Action* associated to it.
- Every tasks must be delivered, then for each vehicle, the combined set of their *TaskAction* lists included in the *VehiclePlan* structure must be equal to twice the set of all tasks as for each task there are two actions.
- All along its *VehiclePlan*, a vehicle load can not exceed its maximal capacity.

1.3 Objective function

The goal of our optimisation is to maximise the reward for an agent. As we said previously, the final state is reached once every tasks has been delivered. However, as task reward is always constant, the total amount of reward remains constant regardless the *VehiclePlans* we use. We therefore want to find the plan that minimized the overall cost (among all vehicles). It's computed in two parts, first the cost of the journey from the vehicle's initial position to the position of it's firts task, then the cost of the trip for between each task and the next one. In the following equation, $\mathcal{V}$ represents the set of all vehicles, $\mathcal{T}_\mathcal{V}$ the ordered list *TaskAction* for vehicle v.

$$\min_{v\in\mathcal{V}, t\in\mathcal{T}} \sum_{v\in\mathcal{V}} \texttt{COSTperKM}(v)\cdot\texttt{DIST}(initCity(v), firstPickup(v))$$

$$+\sum_{v\in\mathcal{V}}\sum_{t_v\in\mathcal{T}_\mathcal{V}} \texttt{COSTperKM}(v)\cdot\texttt{DIST}(currCity, nextCity)\Big|_{currCity(0)=initCity(v),\ currCity=nextCity}^{nextCity=\Big|_{\text{deliverCity}(t_v)\text{ else}}^{\text{pickupCity}(t_v)\text{ if }t_v\ isPickup}}$$

2 Stochastic optimization

2.1 Initial solution

We tested two different ways to find our first solution. First, we assign all tasks to the vehicles with the largest capacity. In this way, if a task has a load greater than the largest of the vehicles' capacities, we know directly that the problem is not realizable. By testing this initial solution we observed that a single vehicle was sufficient to deliver packages optimally. Then, we tried to distribute the tasks randomly so that each vehicle initially had the same number of tasks, always respecting the maximum capacity of the vehicle. It was then noticed that this was not a good initial solution since the overall cost was much less optimized than with the first approach. We can therefore deduce that the success of the problem is very linked to its initialization.

2.2 Generating neighbours

To generate the neighbours of a particular solution, we choose a random vehicle such that it has available tasks to do, and then we apply the two stochastic operators mentioned in the statement. First, the *changingVehicle* method take the first task from tasks of the not empty vehicle and give it to another (possibly an empty one). In this way, a problem can be initialized with only one vehicle carrying the tasks and at the end of the optimization have several in service. Note that when we remove a task, we remove both pickup and deliver actions. Similarly, when we add it to another list, we first add the pickup action and then the deliver action. The second method *changingTaskOrder* returns a set of neighbours obtained by reorganizing the *TaskActions* in the *VehiclePlan* of each vehicle. The reorganization is actually a permutation of all possible combinations of *TaskActions*. As always, this reordering respects the constraints of the problem. The way to generate neighbours in metaheuristic optimization is the key to the problem. A bad neighbourhood generation of neighbors will lead to bad results in optimization.

2.3 Stochastic optimization algorithm

For the optimization algorithm itself, we tried two different approach. In both cases, we generate neighbours at each iteration and with a given probability, we take the neighbour which has the smallest cost as the new solution. First, we implemented an optimization based on the principles of **Simulated Annealing**, a basic metaheuristic algorithm. An important point in Simulated Annealing is cooling: a temperature variable (initially high) gradually decreases. This decrease in temperature can be interpreted as a slow decrease in the probability of accepting worse solutions than the current one as the solution space is explored. By comparing the neighboring solution to the current solution, the algorithm accepts or not to move from to the neighbor point. This decision depends on whether the neighbouring solution is better than the current solution, in which case, the acceptance probability is always equal to 1. Otherwise, the algorithm makes the move with a deacreasing probability of $\exp(\frac{\text{currentSol}-\text{neighbSol}}{\text{temperature}})$. However, in our problem, our way of generating neighbouring points is problematic. Several task permutations result in the same cost, although the exchanged tasks do not necessarily have the same departure or arrival cities. Thus, the difference between the old solution and the best new one (from *chooseNeighbours*) is very often 0, which gives an acceptance probability equal to 1 too often. It gets us stuck in local minima.

We therefore followed the algorithm proposed in the statement. However, we have made some changes. We always accept a new best solution contrary to the statement. On top of that, to avoid getting stuck in local minima and wasting computation time, we have brought equality between two tasks permutations such that the 2 swapped tasks have the same pickupCity (since their totalCost remains the same). Finally, each time a neighbouring solution is computed but is not optimal, we keep it in memory so that it is not included in the new neighbours again. In this way, we always explore new solutions !

3 Results

3.1 Experiment 1: Model parameters

3.1.1 Setting

Topology: England - Task configuration from template - Tasks: 30 - Vehicles: 4 - seed: 12345. We vary the acceptance probability of a new solution from 0 to 1.

3.1.2 Observations

For this experiment, we reduced the calculation time to reach the optimal solution. We achieve an optimal solution by running until timeout around 11400, but here we just want to observe the behaviors of the algorithm under different parameters.

It may also be noted that it would have been wise (and with a little more time) to do not make only one measurement for each probability. As these are non-deterministic results, an average of several samples would have resulted in more accurate results. It can be seen that the most optimal costs are most quickly around 0.4. On the contrary, when the probability of acceptance is close to 1, i.e. when we always accept the best neighbours, we notice that more solutions are blocked in local minima. This is the case with an acceptance probability of 1 where we arrived at a local minimum around 18000, without ever having been able to go there. When the probability tends towards 0, we accept neighbours only if it is better than the best point so far. We observe that a local minimum has stuck us here again.

Acceptance Proba	0	0,2	0,4	0,6	0,8	1
Best Solution <45K ComputationTime[ms]	9682	9167	9264	7788	9999	8505
Best Solution <35K ComputationTime[ms]	12705	11163	12063	10143	11242	10269
Best Solution <25K ComputationTime[ms]	15988	16080	17379	14229	14902	17074
Best Solution <15K ComputationTime[ms]	-	210292	91723	181981	87724	-

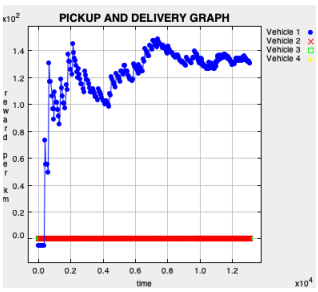
3.2 Experiment 2: Different configurations

3.2.1 Setting

Topology: England - P=0.4 • 0) Tasks: 10 - Vehicles: 6 • 1) Tasks: 10 - Vehicles: 4 • 2) Tasks: 30 - Vehicles: 4 • 3) Tasks: 40 - Vehicles: 4 • 4) Tasks: 60 - Vehicles: 4

3.2.2 Observations

Every time we see that only one vehicle delivers the packages. This can be explained by the fact that the company's efficiency is not part of optimization. As only the distance travelled is relevant, the number of cars in circulation does not greatly influence the final result. Then, we can say that optimal plans are not fair, some vehicles do all the work, and others do nothing.



Not Optimal 4^{th} experiment

The most difficult task is to find all possible combinations of actions in a *VehiclePlan*, and it takes time $O(n^3)$, where n is the number of tasks in the plan. In order to illustrate this point, for our for 5 runs 0 to 4, we were able to call *chooseNeighbours* function, respectively 13255, 14269, 880, 203, and only 4 times until the timeout. As we can see the complexity of the algorithm is almost constant with the number of vehicles. The small difference is due to the use of the *changingVehicle* function, which is more expensive when the number of vehicles increases. This method is accessed v times, where v is the number of vehicles. Since this method itself is O(1), the complexity of the algorithm is then $O(n^3 + v) = O(n^3)$.