Excercise 4
Implementing a centralized agent

Group №67 : Samuel THIRION, Jonas SCHWEIZER

November 3, 2020

1 Solution Representation

The reflexion is based on the CSP model given for the particular case when a vehicle can carry only one task at a time. The main idea is the following : a task is in fact the combination of two subtasks, called `TaskStep` : the pickup and the delivery. In the case of *one carried task per vehicle at a time*, the pickup and delivery subtasks of the same task do always follow each other in this order in a vehicle. Generalizing the model to *several carried task per vehicle at a time* simply means that the pickup and delivery subtasks do not have to be subsequent anymore.

For n tasks on the map, there are in fact $2n$ `TaskStep` (subtasks) that should be assigned in the CSP.

1.1 Variables

The variables are described with the class `VariableSet`. It contains only $X = \{nextTaskV, nextTaskT, time, vehicle\}$ where $nextTaskV(v)$ yields the first `TaskStep` of the vehicle v (can be only a pickup subtask), and $nextTaskT(ts)$ yields the next `TaskStep` to execute after having carried out ts. It is guaranteed that a `VariableSet` instance is a correct solution, as constraints and domains satisfactions are checked upon the generation of the instance (see 2.1 and 2.2).

It is to be noted that a `TaskStep`, is linked to a single city (city of pickup or city of delivery), which is useful for cost computation.

1.2 Constraints

To the contraints described in the assignement, the following ones are added and checked upon generation :

- For a given task T assigned to a vehicle v (i.e. its two subtasks $pickup_T$ and $deliver_T$ are planned for the vehicle v, then $time(pickup_T) < time(deliver_T)$. Tasks have to be picked up before being delivered.
- At any time, a vehicle do not carry more than its capacity (the sum of all $pickup_T$ which associated $deliver$ have not been executed is smaller than the capacity of the vehicle).

1.3 Objective function

The objective function is as described in the assignement : the total cost for the company. That is, $C = \sum_{v \in vehicles} totalDistance(v) \cdot cost(v)$ and

$$\forall v \in vehicles, totalDistance(v) = dist(v.currentCity, nextTaskV(v).city) + \sum_{ts \in v} dist(ts.city, nextTaskT(ts).city$$

Where ts is the subtasks (`TaskStep`) in v and $dist(city, NULL) = 0$. It should be noted that given the representation of the problem in `VariableSet`, it is fairly easy to compute all the tasks planned for a given vehicle.

2 Stochastic optimization

2.1 Initial solution

The initial solution is simple : all `TaskStep` pair of the same task (pickup and delivery) are assigned one after the other, in this order, to the vehicle whose capacity is the greatest. This is equivalent to say "*the biggest vehicle will pickup and deliver all tasks, with one task on board at a time*".

This solution is not random as recommended by the assignement, but seems not to affect the performance of the model. To pick a *more* randomized initial solution, an easy trick would be to pick a random neighbour of the previous solution found, and repeat the process n times. After a while, the initial solution would be significantly different from the first described choice.

2.2 Generating neighbours

Here are the two ways of generating the neighbours of a given solution A_{old} . There are implemented `ChooseNeighbours()` of the static class `CPMaker`.

Task changing vehicle : after having randomly chosen a vehicle with at least a task planned for it, we remove the first `TaskStep` (and its sibling related to the same task), and place this just-extracted {pickup T, deliver T} `TaskStep` duo in front of another vehicle's plan. This leads to $|vehicles|$ new neighbours. Through various experiments, it has been noticed that the model could get stucked to a very unpleasant local minimum[1]. To escape these types of local minima, we perform again the latter operation on all the generated neighbours, and again and again for a total of `DEPTH_SEARCH` number of times. This allows to potentially remove several tasks of a vehicle in one step, and escape a minimum described in the footnotes.

Changing tasks orders inside a vehicle : after having selected a vehicle with at least a task planned for it, neighbours are generated this way : the algorithm tries to exchange every subtask with another inside the plan of the vehicle. Each time this exchange do not violate any constraint, this new `VariableSet` is added to the neighbours of A_{old} .

2.3 Stochastic optimization algorithm

In order to optimize the search of the minimum, we needed to choose some parameters. After that the minimum cost neighboring state (MCNS) has been found, we draw a random number between 0 and 1. The value of this number determines what we choose for the next state.

- $]0, \texttt{takeBestThreshold}]$: next state is simply the MCNS
- $]\texttt{takeBestThreshold}, 1 - \texttt{explorationThreshold}]$: next state is the current state (we take none of the calculated neighboring states).
- $]1 - \texttt{explorationThreshold}, 1]$: next state is chosen randomly in all the neighboring states calculated. Nonetheless, we save the current local minimum in case this was the global one.

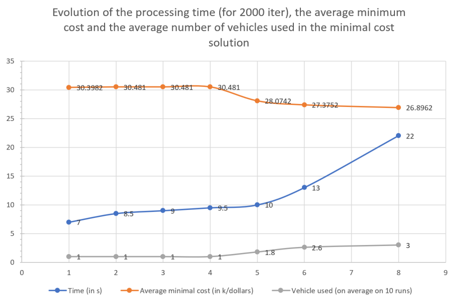
[1]Notation : $City_x(pickup_y)$ is the subtask *pickup* of the task y, that takes place in $City_x$
If v_1 hast a cost of $100\$/km$, v_2 has a cost of $1\$/km$ and the current solution is $\{v_1 \rightarrow City_1(pickup_1) \rightarrow City_1(pickup_2) \rightarrow City_2(deliver_2) \rightarrow City_2(deliver_1)$, $v_2 \rightarrow NULL$ $\}$, then changing the task 1 from v_1 to v_2 will increase the overall cost (nor will reorder tasks inside v_1). To escape this local minimum, both task 1 and task 2 have to be removed from v_1

`takeBestThreshold` and `explorationThreshold` are first set to default values and then variate according to the characteristics of the MCNS. For instance, the `takeBestThreshold` is increased if the MCNS is a lot better than the previous state (cost difference is big), or if the number of used vehicles is lower [2]. The `explorationThreshold`, for instance, is increased if we are stuck (cost of MCNS and current state are the same, or cost of MCNS even bigger than current state) and reduced when MCNS is better than the current state. Therefore, the search tends to last longer as the cost of new states decreases, and tends to explore new grounds if stuck in a local minimum.

3 Results

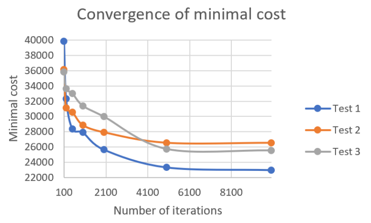
3.1 Experiment 1: Model parameters

3.1.1 Depth



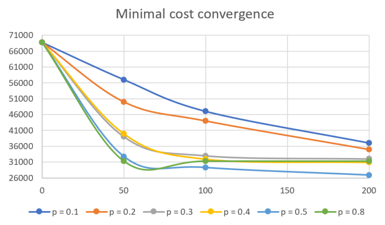
The `depth` parameter allows us to get out of unfortunate local minimums. On the figure, there are 30 tasks and we see that if `depth` $<= 4$, we can't distribute the tasks on more than 1 vehicle. The greater the `depth`, the less frequently you get stuck in a local minimum but the greater is the calculation time. `depth` $= 8$ is a good compromise between avoiding getting stuck and processing speed (for 30 tasks).
Fig : x-axis = depth parameter

3.1.2 Number of iterations



We see that the more iterations, the longer to process but the lower minimal cost. We also see that after 5000 iterations, we rarely improve the cost (because stuck in local minimum). This maximum optimal number of iterations is for 30 tasks.

3.1.3 takeBestThreshold



`takeBestThreshold` is a parameter to choose wether you take the best neighbor or you keep the previous version. The figure shows the convergence of minimal cost for different values of `takeBestThreshold`: we see that the bigger, the faster the convergence, as expected.
Fig : x-axis = Number of iterations

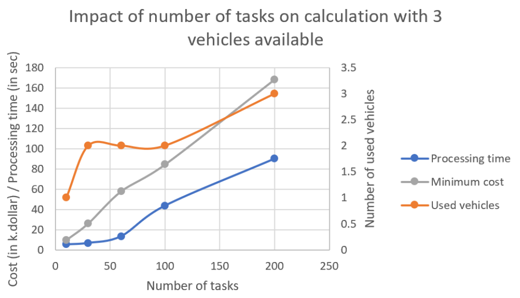
3.1.4 Improving takeBestThreshold and adding exploration

In order to improve convergence towards interesting states, we can choose to adapt this threshold depending on the best neighbor found (see 2.3). In average on 10 tries, customizing this value with the best neighbor can improve by 5% the estimated minimal cost at a given number of iterations. To avoid being stuck in a local minimum, we add a possibility to explore when the cost is stable. For a same number of iterations (2000), adding exploration can reduce the minimal cost found by 17% (26k to 22k with `explorationThreshold` to 0.005 or 0.01).

3.2 Experiment 2: Different configurations

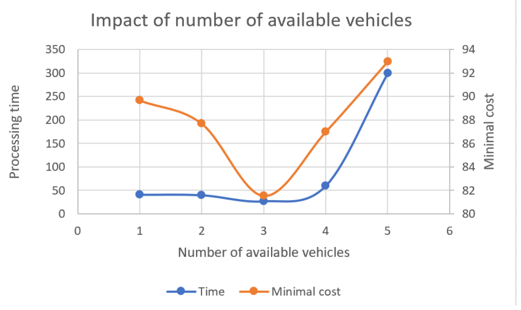
3.2.1 Number of tasks

[2]We found that with a small number of tasks, using less vehicles is more optimized



The processing time is linear in the number of tasks. As the number of tasks increases, the number of used vehicles also increases. However, we see that the convergence toward the optimal cost is slower for a large amount of tasks: for a fixed number of iterations, the minimal cost found is linear, but with more tasks we can further improve this minimum by running with more iterations. (with nTasks=30 nIter=2000: min=25000 and for nIter=10000: min=23000 while with nTasks=200, nIter=2000: min=166000 and nIter=10000: min=135000).

3.2.2 Number of vehicles



The number of available vehicles is tricky. Our algorithm optimises everything according to the distance, so even with 2 vehicles, there is always a vehicle that goes almost everywhere, so adding new ones only improves the amount of neighbors generated at each step, increasing the processing time. Situation is different for 1 to 3 vehicles, because for 3, the average number of neighbors generated is less than for 1 vehicle (because you have 2 chances out of 3 to be on a vehicle that does not have a lot of tasks). The convergence speed descreases with the number of vehicles available.