Excercise 4
Implementing a centralized agent

Group №27 : Carolin Beer, Francesco Pase

December 13, 2018

1 Solution Representation

1.1 Variables

Solutions are coded in Solution objects containing an array of ArrayLists of Tuplas. Every entry in the array is an ArrayList representing the plan of a vehicle. ArrayLists contain Tupla objects coding actions in four variables: *task* is the Task implied in the action; *action* as int value (1 for pick up, 2 for deliver); *capacityLeft* is the remaining capacity of the vehicle after current action; *cost* is the cumulative cost of the plan until the end of the current action.

1.2 Constraints

The method *checkMove()* check for constraints when one action is changed. Logically constraints: in the ArrayLists of the plans, the pick up Tupla of one action can not appear after its related deivery Tupla while delivery Tuplas must be placed after the pick up one. Capacity constraints: the variables *capacityLeft* of every Tupla in every plan must be a value between 0 and the initial vehicle capacity.

1.3 Objective function

The function to be optimized is the sum of every vehicle's cumulative cost ($km \times costperkm$). It is computed as: $\sum_{v \in Vehicles} v.get(v.size() - 1).cost$, where v is an entry in the array representing the plan for v (ArrayList) and its last entry contains its final cumulative cost.

2 Stochastic optimization

2.1 Initial solution

All tasks are assigned sequentially to the vehicle with the highest capacity.

2.2 Generating neighbours

Starting from a temporary solution we apply two transformations to generate neighbors: (1) We randomly choose one vehicle, take one task at random and move it to the end position of every other vehicle's plan (moving its pick up and delivery Tuplas at the end of the new vehicle's List). (2) For every *newSolution*, we take the vehicle *v1* from which the task was removed and, for every action in its list, we take it and try to place it in every other correct position generating other neighbors. (3) Go back to 1 one more time.

2.3 Stochastic optimization algorithm

We implement a Stochastic Local Search algorithm. Our model has 2 parameters: *currentProb* and *jumpWhen*. At every iteration *search()* generates new neighbors and performs local search calling *localSearch()* that behaves like this: it finds a random and best solution among neighbors; If the best is better (in terms of cost) than the temporary one, it selects this one as the new temporary; if not, take anyway the best neighbor as new temporary solution with probability *currentProb* (*stuck*++); otherwise, jump to a random neighbor if $stuck > jumpWhen$ (this is used to deterministically escape from stuck points) and set $stuck = 0$. The method *search()* then updates the parameters: in the first phase, *currentProb* is set to 0.8 and *jumpWhen* to 25. After one third of the total time (timeout time) the second phase starts: $currentProb = 0.6$ and $jumpWhen = 50$. After two-third of available time, the third phase starts: $currentProb = 0.3$ and $jumpWhen = 75$.

3 Results

3.1 Experiment 1: Model parameters

3.1.1 Setting

England as topology, 30 tasks, 4 vehicles and different values of *currentProb* and *jumpWhen* were tried. Such experiments were performed in order to heuristically tune the values of the parameters so to find the ones that lead to best performance in terms of solution cost and convergence time.

3.1.2 Observations

As expected, using high value of *currentProb* and low values for *jumpWhen* helped to explore new solutions. By adding more variability, the model falls in local minima with lower probability but the convergence of the algorithm toward low values of cost function was very slow.

3.2 Experiment 2: Different configurations

3.2.1 Setting

Topology: England. Different numbers of tasks and vehicles were used as well as different timeouts. Task configurations were kept at default values. The timeout in the second table was fixed at 1 min. for every simulation.

3.2.2 Observations

#Tasks	Vehicles	Initial Cost	Final Cost (5min)	Final Cost (1min)	Final Cost (20 sec)
30	3	68'731	15'223	16'600	16'741.5
30	4	68'731	14'737	14'800	18'542
30	5	68'731	16'658	16'900	17'066.5

#Tasks	Initial Cost	5 Vehicles Final Cost	4 Vehicles Final Cost	3 Vehicles Final Cost
20	44'214	13'369	12'654	11'267
50	123'297	29'308	29'854	29'150

At every iteration the algorithm is $\mathcal{O}(\mathcal{T}^2 \times \mathcal{V})$ ($\mathcal{T}$ = # tasks and $\mathcal{V}$ = # of vehicles). We can see that the more time we give to the model, the better the results, as expected. Moreover, adding vehicles not always helps. Indeed the function to optimize is the total cost to deliver all tasks (not total time) and so usually the best solution makes some vehicles deliver more tasks than the others (just because of their paths). Given that, with few tasks adding new vehicles just adds more complexity and more solutions have to be explored so the convergence is slower (e.g. with small amount of time, using just 3 vehicles leads to the best solution). For example, in this simulation 2 vehicles take the majority of the tasks (Topology England, 4 vehicles, 30 tasks, timeout 30s). The cost of the solution is 16611.5 and the solution is:
9 24 11 15 24 15 7 29 16 2 23 16 23 21 29 11 7 9 21 2
8 14 14 8
17 28 10 12 19 26 1 28 12 19 10 3 1 18 26 18 17 25 0 3 0 20 20 25
27 4 6 22 4 6 5 22 5 13 13 27
Every line is a plan for a vehicle and every number represents the task id (first occurrence is the pick up action, second the delivery).