Excercise 4
Implementing a centralized agent

Group №90: Ruben Janssens, David Resin

November 5, 2019

1 Solution Representation

1.1 Variables

The N_T tasks are represented in our implementation by integers t_i with $i \in [0, N_T)$, and the N_V vehicles are represented by integers v_j with $j \in [0, N_V)$.

The solution is then represented by the array $orders$. This is an array which contains N_V lists of variable length. Each list $orders_{v_j}$ can contain 0 to $2N_T$ elements, which are all integers in the interval $[0, N_T)$. These integers represent the pickup and delivery of different tasks, performed by vehicle v_j. The vehicle will automatically move to the city required for the next pickup or delivery. The first occurrence of a task is always interpreted as the pickup, the second occurrence as the delivery.

1.2 Constraints

The solution has the respect the following constraints. Every task has to be picked up: $\forall t_i : \exists v_j : t_i \in orders_{v_j}$. Every task that is picked up, also has to be delivered: $\forall t_i \in orders(v_j) : occurrences(t_i, orders_{v_j}) \geq 2$ (with $occurrences(t_i, orders_{v_j})$ representing the number of occurrences of t_i in $orders_{v_j}$). Every task can only be handled by one vehicle: $t_i \in orders_{v_j} \Rightarrow \forall v_k \neq v_j : t_i \notin orders(v_k)$. The total weight of the tasks any vehicle is carrying at any point in its journey cannot exceed the capacity of the vehicle.

1.3 Objective function

The function that is optimized is the total cost incurred by all the vehicles. It is computed as follows. For every vehicle, it computes $cost(v_i) = \sum_{j=0}^{length(orders_{v_i})-1} cost\_per\_km(v_i) \cdot distance(orders_{v_i}(j), orders_{v_i}(j+1))$ with $cost\_per\_km(v_i)$ being the cost for every kilometer travelled by vehicle v_i, and $distance(t_i, t_j)$ being the distance in km the vehicle has to travel between the city where task t_i has to be picked up or delivered, and the city where task t_j has to be picked up or delivered (pickup or delivery depending on whether it is the first or second occurrance in $orders_{v_i}$). The algorithm then searches for the solution with the lowest $total\_cost = \sum_{v_i} cost(v_i)$.

2 Stochastic optimization

2.1 Initial solution

The initial solution is a random distribution of the tasks over the vehicles.

This solution is generated by first adding each task $t_i \in [0, N_T)$ to $orders_{v_j}$ for a random $v_j \in [0, N_V]$ twice, once for pickup and once for delivery. Then, every list $orders_{v_j}$ is shuffled to create a random order in which the tasks are picked up and delivered.

A random initial solution was chosen because adding all tasks to one vehicle (in a strict pickup-delivery-pickup-delivery order), like in the algorithm given, often led to the algorithm not being able to climb out of the local minimum of this initial solution.

2.2 Generating neighbours

The algorithm starts by picking a random vehicle v_j for which $length(orders_{v_j}) > 0$. Neighbours are generated in four ways. For every vehicle $v_k \neq v_j$, generate a neighbour by moving the first task t_i in $orders_{v_j}$ to v_k by removing both occurrences of t_i from $orders_{v_j}$ and adding t_i twice (once for pickup, once for delivery) at the beginning of $orders_{v_k}$. If $length(orders_{v_j}) > 2$, generate a neighbour for every pair of two elements in $orders_{v_j}$ by swapping the two elements, as long as the pair represents two different tasks. Shuffle all elements in $orders_{v_j}$ randomly. Shuffle a random number of tasks towards other vehicles. This is done by randomly choosing a random number of tasks in $orders_{v_j}$, removing both occurrences of each task from $orders_{v_j}$ and placing each task at two random locations in $orders_{v_k}$ for a random $v_k \neq v_j$.

Every neighbour that is generated in these ways and that satisfies the constraints, is then added to the neighbours list. If no valid neighbours were found, the current solution is returned. Because of the randomness included in finding the neighbours, it is possible to find a valid neighbour in the next iteration.

2.3 Stochastic optimization algorithm

The basic stochastic optimization algorithm follows the same flow as the algorithm given in the assignment. It starts by generating an initial solution as described in section 2.1. It then generates all valid neighbours using the methods described in section 2.2. From these neighbours, it chooses the one with the lowest *total_cost*. Then, it either returns the current solution (with a probability of *proba_random*, which is a parameter of the model), or returns that neighbour with the lowest cost. If there are multiple neighbours with the lowest cost, it chooses one at random. This process is repeated for a fixed number of iterations.

However, we saw that this algorithm is very dependent on the initial solution and often does not leave the local minimum around the initial solution. We therefore extended the algorithm to consider multiple initial solutions and so perform a search in solution space that is not purely depth-first. The algorithm takes two extra parameters for this procedure: *num_stages* and *num_iterations*. We think this is very important for the given approach, as results tend to vary greatly and it is just not good enough in our opinion. Having variance from simple to double in some rare cases is quite extreme.

This extended algorithm works in a tree structure. It starts by considering 2^{num_stages} initial solutions. It then improves each of these initial solutions for $num\_iterations/2^{num\_stages-1}$ iterations, like in the basic algorithm described above. Then, the $2^{num\_stages-1}$ best solutions are chosen. These solutions are then again improved for $num\_iterations/2^{num\_stages-2}$ iterations, after which the $2^{num\_stages-2}$ best solutions are chosen, and so on until one solution remains..

3 Results

3.1 Experiment 1: Model parameters

3.1.1 Setting

In the following experiments, we run our tree algorithm with 8 stages and 2000 iterations. Through our experimenting we deduced that these two parameters had among the best and most consistent results while still being rather fast. To obtain them, we did a lot of testing with different values and triangulated our way to 8 and 2000 which provided consistently good results. 8 means we start with 256 initial solutions, we feel it's an appropriate balance as this seems quite probable to contain good starting points.

We chose 2000 iterations because more didn't seem to improve anything while less clearly did. Again, trying to not have too long of a computation time as well.

Here we are observing what happens when we vary the p value. We test the values by increments of .2 from .1 to .9. The map is England with 4 vehicles with default parameters.

3.1.2 Observations

We clearly see that a low p value is favourable with a lower cost. It seems to indicate that returning the current assignment is just slowing down the process and nothing else, which is what we understood. Please note that our p is reversed compared to the original algorithm, which means in the definition given in the assignment, a high p value is favourable. Here are the average values of our experiments:

p value	.1	.3	.5	.7	.9
Average cost	17063	17515	20207	22192	24415

3.2 Experiment 2: Different configurations

3.2.1 Setting

Three parameters of the configuration were varied: the number of tasks, the number of vehicles and the vehicle capacities. The number of tasks was varied between 20, 30 and 40, with 4 vehicles which all have capacity 30. The number of vehicles was varied between 2, 3 and 4, all with capacity 30 and with 30 tasks. For the vehicle capacities, one experiment was ran with all 4 vehicles having capacity 30, and another with capacities 10, 20, 30 and 40. All experiments were run on the England topology, with $proba\_random = 0.3$, $num\_stages = 8$ and $num\_iterations = 2000$, and the task distribution is the same as in the default configuration. For every experiment, the mean total cost out of three runs is given, togehter with the mean computation time (in ms).

3.2.2 Observations

# Tasks	Total cost	Time	#Vehicles	Total cost	Time	Capacities	Total cost	Time
20	12964	19554	2	16112	138147	all 30	19329	63824
30	19650	52943	3	17886	80446	10, 20, 30, 40	20494	62562
40	25700	120631	4	19329	63824			

For the number of tasks: the total cost goes up with a higher number of tasks. This is logical, since a larger distance will have to be travelled. The computation time also rises steeply however. This originates from the neighbour generation algorithm, which will generate a new neighbour for every pair of tasks.

A rising number of vehicles also comes with a (slightly) higher total cost. However, the computation time actually decrease steeply: that seems to indicate that while the first and fourth neighbour generation algorithms are dependent on the number of vehicles, the second neighbour algorithm will dominate, because the tasks are more spread out over the vehicles, the total number of tasks per vehicle is smaller, so the number of pairs is smaller as well. It seems that the algorithm can get to a lower cost for 2 vehicles, because it explores many more solutions.

Changing the capacities of the vehicles does not seem to have a big effect on the computation time. The cost is slightly higher for the distributed capacities, which could be explained by the total capacity of all vehicle, which is lower, so the vehicles can't carry as many tasks at once.

The tasks are in almost no case distributed evenly over all vehicles. In the experiments with 4 vehicles, often two vehicles get a large amount of tasks (around 10 or more in the case of 30 tasks), while others get less (0 to 5). The total cost function does not take into account the time the delivery routes take, which explains this behaviour.