Excercise 4
Implementing a centralized agent

Group №23: Jean-Thomas Furrer, Emily Hentgen

November 7, 2017

1 Solution Representation

1.1 Variables

We use two variables for representing a plan resulting from the Stochastic Local Search (SLS) algorithm:

Map<Vehicle, TaskAction> *vehicleToFirstTaskAction*: maps a vehicle to the first task action it has to perform (necessarily a pick up, may be `null` if the vehicle picks up no task at all)

Map<TaskAction, TaskAction> *taskActionToTaskAction*: maps a task action to the next task action performed by the same vehicle; used jointly with *vehicleToFirstTaskAction* in order to know which vehicle handles which set of tasks

1.2 Constraints

We (explicitly) enforce 2 constraints:

- The load of each vehicle must not exceed its capacity.
- The delivery of a task cannot happen before the pickup of this same task.

Otherwise, constraints such as 'all tasks must be delivered' are enforced by the design of our SLS algorithm.

1.3 Objective function

The objective function we optimize is the total travel cost resulting from all pickups and deliveries of all vehicles.

2 Stochastic optimization

2.1 Initial solution

One possible initial solution consists of assigning all tasks to the vehicle with the largest capacity, in any order, if possible: if the weight of the heaviest task exceeds the capacity of the largest vehicle, then there is no solution to this pick up and delivery problem. In *vehicleToFirstTaskAction*, we add an entry mapping the largest vehicle to a any task pickup, and map all the other vehicles to the `null` value. In *taskActionToTaskAction*, we begin by mapping the first task pickup to its corresponding task delivery. We then map this task delivery to any the pickup of any task in the set of remaining tasks, and map the latter task pickup to its corresponding task delivery, and so on. Each time a task is selected, we remove it from the set of remaining tasks: the mapping *taskActionToTaskAction* is complete once the set of remaining tasks is empty.

2.2 Generating neighbours

We use almost the same two transformations as in the Pickup and Delivery Problem where each vehicle is allowed to carry only one task at a time: a neighbour plan can be generated by either swapping the order of two tasks for the same vehicle, or by giving the first task of one vehicle to another. We firstly generate a potential neighbour plan without enforcing the constraints. It is after it has been generated that we check whether the constraints are respected: if this is the case, we add it to the set of neighbouring plans, otherwise, we discard it.

2.3 Stochastic optimization algorithm

Our stochastic optimization algorithm will stop either because it reaches the timeout limit defined at the beginning of the simulation, or because there was no improvement in the cost of the plan for a certain number of iterations. For selecting the next plan, it uses a local choice function which, depending on a certain probability distribution, returns a neighbouring minimal cost plan, the current plan, or a random plan (which can be a minimal cost plan). This is because systematically returning a minimal cost plan may lead the stochastic local search to get stuck in a local minimum. This also depends on the initial solution defined at the setup.

Algorithm 1 Stochastic Local Search

```
INPUT
    List<Vehicle> vehicles                          ▷ a list of the agent's vehicles
    TaskSet tasks                                   ▷ the set of tasks to deliver
OUTPUT
    Plan plan                                       ▷ a (sub)optimal plan

initialCountdown ← 100 000, countdown ← initialCountdown
hasTimedOut ← False, noImprovement ← False
minimumCostAchieved ← ∞
plan ← selectInitialSolution(tasks), bestPlan ← null

while ((not hasTimedOut) and (not noImprovement)) do
    previousPlan ← plan
    neighbourPlans ← chooseNeighbours(previousPlan)   ▷ returns the neighbours of the current plan
    plan ← localChoice(neighbourPlans)                ▷ selects the next plan

    if (duration ≥ timeout) then                      ▷ in practice, subtract a margin to the timeout
        timeOut ← True                                ▷ (e.g. 1s) so that the algorithm stops before it
    end if                                            ▷ effectively times out
    if (plan.cost >= minimumCostAchieved) then
        − − countdown
    else
        countdown ← initialCountdown
        minimumCostAchieved ← plan.cost
        bestPlan ← plan
    end if
    if (countdown = 0) then
        noImprovement ← True
    end if
end while
return bestPlan
```

3 Results

3.1 Experiment 1: Model parameters

Our SLS algorithm has three parameters: the probability that a minimum cost plan is selected, the probability that a random plan is selected, and the maximum number of iterations without improvement before the search stops.

3.1.1 Setting

We mostly tested the performance of the SLS algorithm for different probability distributions. If the maximum number of iterations is "large enough", but not "too large", the SLS algorithm often manages to find a (sub)optimal plan before it actually times out (which happens after 303 000 ms). A value of 10 000 seems reasonable: the algorithm does not stop too early, and still manages too find a plan with a cost consistent throughout multiple simulations. For these simulations, we kept the initial seed of 12345.

3.1.2 Observations

We report the minimum observed cost, maximum observed cost and average cost of the suboptimal plans, as well as the average execution time, over roughly 6 simulations per configuration.

number of tasks	20				30			
$p_{minimum\ cost\ plan}$	0.3	0.4	0.5	0.7	0.3	0.4	0.5	0.7
$p_{current\ plan}$	0.5	0.4	0.3	0.1	0.5	0.4	0.3	0.1
$p_{random\ plan}$	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2
minimum cost observed	9 347	9 347	9 347	9 347	13 484	12 876	12 451	11 559
maximum cost observed	10 072	10 679	11 931	11 429	16 693	15 144	14 492	15 392
average cost	9 627	9 894	10 246	10 248	14 325	13 715	13 567	13 382
average execution time (ms)	95 207	95 774	71 630	86 475	161 027	227 860	200 463	209 973

Table 1: seed: 12345, countdown: 10 000, initial solution: assigns all tasks to the largest vehicle

For a relatively large probability of picking a minimum cost plan, most of the time, the plan found repeatedly reaches a low cost, particularly for 20 tasks. Some plans however turn out to have a somewhat larger cost, which confirms our intuition about getting stuck in a local minimum. For 30 tasks, the execution time is significantly longer, and the different probability distributions to not systematically find the lowest cost plan. In fact, the SLS algorithm sometimes times out, which is most likely because the complexity of the problem is now much larger than before.

3.2 Experiment 2: Different configurations

We observed that for a large number of tasks (e.g. 20 or more), the SLS algorithm tends to assign all tasks to a single vehicle (the vehicle that initially gets all the tasks). There may be two reasons for this. One may be that the number of neighbouring plans where the order of the tasks is changed is much larger than the number of neighbouring plans giving the first task of a vehicle to another one. The other may be that it is simply cheaper and easily feasible to have a unique vehicle doing all the work, since there are many tasks to deliver, and only few cities: a large enough vehicle capacity combined with tasks of small weights gives the vehicle a certain flexibility for the order in which tasks can be picked up and delivered - a flexibility which is not necessarily larger when there are more vehicles involved. In this experiment, we want to compare the performance of the SLS algorithm for a different initial solution.

3.2.1 Setting

We kept the same setting as in Experiment 1, except for the initial solution: at the beginning, each vehicle now receives roughly the same number of tasks, as long as it can carry each one of them.

3.2.2 Observations

number of tasks	20				30			
$p_{minimum\ cost\ plan}$	0.3	0.4	0.5	0.7	0.3	0.4	0.5	0.7
$p_{current\ plan}$	0.5	0.4	0.3	0.1	0.5	0.4	0.3	0.1
$p_{random\ plan}$	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2
minimum cost observed	10 480	9 889	13 042	9 347	12 543	13 004	13 486	18 036
maximum cost observed	14 557	14 199	16 076	13 542	19 882	21 930	19 209	23 408
average cost	12 014	14 718	11 918	12 225	16 405	18 203	17 404	19 897
average execution time (ms)	43 634	21 244	63 120	58 786	182 548	88 736	87 360	62 573

Table 2: seed: 12345, countdown: 10 000, initial solution: assigns roughly an equal number of tasks to all vehicles

With this different initial solution, the search stops sooner on average, and the resulting cost is higher. If we were to increase the countdown, this initial solution might eventually display lower plan costs though, closer to the costs of the plans found with the previous initial solution. What is more, the SLS algorithm still tends to distribute all tasks between one vehicle, at most two, the others having not task assigned. This suggests that a plan assigning all tasks to a single vehicle is for the company optimal, instead of one using all available vehicles and trying to be "fair" among them all. If we added a time constraint in the simulation - the company's vehicle must deliver the

tasks within a given amount of time, and are penalized if they do not - then the corresponding SLS algorithm would most likely tend to assign the tasks to multiple vehicles instead of only one.