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 that '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 ← 100000, countdown ← initialCountdown
hasTimedOut ← False, noImprovement ← False
minimumCostAchieved ← ∞
plan ← selectInitialSolution(tasks)

while ((not hasTimedOut) and (not noImprovement)) do
    previousPlan ← plan
    neighbourPlans ← chooseNeighbours(previousPlan)
    plan ← localChoice(neighbourPlans)

    if (duration > timeout) then
        timeOut ← True
    end if
    if (plan.cost >= minimumCostAchieved) then
        − − countdown
    else
        countdown ← initialCountdown
        minimumCostAchieved ← plan.cost
    end if
    if (countdown = 0) then
        noImprovement ← True
    end if
end while
return plan
```

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: as long as the number of iterations without improvement is "large enough", its influence on the cost of the plan is going to be limited. We observed that for low values such as 10 000, the SLS algorithm tends to stop because there is no more improvement. Therefore, we set this value to 100 000, such that the algorithm times out for most

simulations (when 303 000 ms is reached): this way, we are guaranteed the search did not stop "too soon". For these simulations, we kept the initial seed of 12345.

3.1.2 Observations

We report the minimum observed cost and maximum observed cost of the suboptimal plans, over roughly 6 simulations per configuration, with a maximum number of iterations without improvement of 100 000.

number of tasks	20				30			
$p_{minimum\ cost\ plan}$	0.3	0.4	0.5	0.8	0.3	0.4	0.5	0.8
$p_{random\ plan}$	0.6	0.3	0.2	0.1	0.6	0.3	0.2	0.1
$p_{current\ plan}$	0.1	0.3	0.3	0.1	0.1	0.3	0.3	0.1
minimum cost observed	9 347	10 249	10 321	9 347	15 226	14 236	14 946	13 062
maximum cost observed	14 201	20 811	22 513	15 593	30 582	37 728	23 646	22 122
average cost observed	11 348	15 897	14 912	11 390	19 513	19 167	17 797	15 971

Table 1: seed: 12345, countdown: 100 000, initial solution: assigns all tasks to the largest vehicle

For a relatively large probability of picking a minimum cost plan, even though most of the time, the plan found has a rather low cost, some of the other plans turned out to have a quite large cost, which confirms our intuition about the likeliness of getting stuck in a local minimum. In average though, setting this probability to a high value, so that the SLS algorithm converges faster, resulted in lower plan costs.
What is interesting as well is that the cost of a plan does not significantly increase from 20 tasks to 30 tasks: the SLS algorithm manages to combine pickups and deliveries so that the cost of a suboptimal plan remains in the same range.

3.2 Experiment 2: Different configurations

We observed that for a large number of tasks, the SLS algorithm tends to assign all tasks, or nearly all tasks, to a single vehicle (the vehicle that initially gets all the tasks), and just one or two to a second vehicle. This may be because 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. 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, only the initial solution has changed: 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	0.8	0.3	0.4	0.5	0.8
$p_{random\ plan}$	0.6	0.3	0.2	0.1	0.6	0.3	0.2	0.1
$p_{current\ plan}$	0.1	0.3	0.3	0.1	0.1	0.3	0.3	0.1
minimum cost observed	9 888	12 871	9 347	9 865	18 041	16 375	15 939	16 914
maximum cost observed	16 696	21 048	24 352	19 072	26 311	38 715	26 591	25 360
average cost observed	13 464	16 112	13 867	14 599	22 778	25 610	20 733	22 469

Table 2: seed: 12345, countdown: 100 000, initial solution: assigns roughly an equal number of tasks to all vehicles

With this different initial solution, the SLS algorithm still tends to distribute all tasks between one or two vehicles. And it turned out that among all these suboptimal plans, the best ones often had only a single vehicle. 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 in they do not - then the corresponding SLS algorithm would most likely tend to assign the tasks to multiple vehicles.