

Modelling for Combinatorial Optimisation (1DL451)

Uppsala University – Autumn 2025

Assignment 3: Scooter Service Problem (SSP)

— Deadline: 13:00 on Friday 12 January 2026 —

The scope of this assignment is Topics 1 to 8: you need *not* show any knowledge of subsequent topics. Read the **Report Instructions** and **Grading Rules** at the end of this document *before* tackling the following problem(s) upon reading them a first time. It is strongly recommended to prepare and attend the help sessions, as huge time savings may ensue.

The objectives of this assignment are (a) to design a model for a problem with a lot of side constraints, (b) to learn how a complex vehicle routing problem can be modelled with suitable global constraints, (c) to observe experimentally the power of large-neighbourhood search (which is a form of local search), and (d) to see the bigger picture, as an optimisation problem is usually just one aspect of a larger problem, here in the sense that the instance data come from a machine-learning pipeline.

Background

Rented electric scooters are an increasingly popular transportation method in urban areas. However, they present new logistic challenges for each company that owns a fleet of electric scooters. We tackle here part of the challenge of deciding how to service the scooters.

We consider scooters of a newer model, where the battery pack can be replaced on location and service employees travel around town with charged batteries in order to replace the batteries of scooters with a too low charge, and possibly do some other minor maintenance. We assume each service employee is paid based on the payment values of the actually serviced scooters.

Since a company need not service all the scooters at once, it can decide every now and then which scooters to service, which employee services which scooters, and which route each service employee travels. Furthermore, when deciding whether or not to service a particular scooter, the company also wants to take into account other factors than its current charge level, such as the likelihood that someone will want to use it soon, its geographical location, and its need for maintenance. For taking these factors into account, one can use methods of modern Data Science in order to create a machine-learning pipeline that, based on historic and current data, generates the matrix of estimated current travel times between any two scooters, as well as predictions (in the form of a scalar per scooter) that represent the service priority and payment value of each scooter.

We consider here what to do once we have such predictions from a machine-learning pipeline: rather than letting the employees decide *individually* for which scooters they grab the intention of service, we use the MiniZinc toolchain in order to make these decisions *automatically* while considering all the employees *collectively*.

The Scooter Service Problem

We are given the following parameters and derived parameters:

- `nScooters` is the number of scooters;
- `nEmployees` is the number of employees;

- `nNodes = nScooters + 2 * nEmployees` is the number of nodes in a directed graph, where the nodes in `1..nScooters` represent the current scooter locations, and the nodes in `(nScooters+1)..nNodes` represent the start and end locations of the employees, as explained below;
- `TravelTime[f,t]` is the estimated time of travelling from node `f` to node `t`, both in `1..nNodes`, *plus* servicing the scooter at `t`, if any; note that `TravelTime` *cannot* be assumed to be symmetric;
- for each scooter `s` in `1..nScooters`:
 - `Priority[s]` is the priority of some employee servicing `s`: a higher value means a higher priority;
 - `Payment[s]` is the payment that any employee would receive for servicing `s`;
- for each employee `e` in `1..nEmployees`:
 - `StartNode[e]` is the start node of `e`;
 - `EndNode[e]` is the end node of `e`;
 - `MinNumScooters[e]` is the minimum number of scooters `e` is willing to service;
 - `MaxNumScooters[e]` is the maximum number of scooters `e` is willing to service;
 - `MaxTime[e]` is the *preferred* maximum total travel time of `e`.

The *scooter service problem (SSP)* is to decide, within the complete directed graph induced by the nodes described above, the route of each employee, which then also determines the scooters that each employee services. The constraints are as follows:

- Each scooter is serviced by at most one employee.
- Each employee `e` in `1..nEmployees` services either 0 scooters (and has an empty route) or from `MinNumScooters[e]` to `MaxNumScooters[e]` scooters.

There are three simultaneous objectives:

- Minimise `totalOvertime`, which is the total overtime of all the employees, where the overtime of employee `e` is the amount by which the total travel time of `e` exceeds `MaxTime[e]`.
- Maximise `totalPayment`, which is the total payment that all the employees receive. Note that this makes sense even from the company point of view: the scooters that have a higher payment value are normally prioritised.
- Minimise `totalPriorityLoss`, which is the total priority of all the scooters that are not serviced by any employee.

In order to combine these objectives into a single objective, we minimise a weighted sum in order to balance them:

```

minimize alpha * totalOvertime
         - beta  * totalPayment
         + gamma * totalPriorityLoss

```

where `alpha`, `beta`, and `gamma` are additional parameters. Note that we do not minimise the total travel time of all the employees: this can be done as a post-processing step where we optimise the route of each employee, but we do not consider doing so in this assignment.

Viewpoint

In order to model the SSP, we use a so-called *giant-tour formulation* to represent the routing of all the employees, where all their routes are merged into a single (sub-)circuit. We more or less transform our vehicle routing problem (VRP) into a travelling salesperson problem (TSP). Formally, we use an array `Succ` of so-called *successor variables*, indexed by all the nodes (the scooters, as well as the employee start and end nodes), where `Succ[n]` denotes the node that is visited after node `n`, and we state a single `subcircuit` constraint on these variables. Note that we use `subcircuit` (and not `circuit`) as some scooters might not be serviced in an optimal solution. For this formulation to be correct, you must state constraints that force the successor of each employee's end node to be the start node of the circularly next employee.

Figure 1 is a sketch of this formulation for three service employees and six scooters that need service: the white and blue squares are respectively the start and end nodes of the employee whose number is written inside the square, while the yellow circles are the scooter nodes. Subfigure 1a shows the additional constraints described above, enforcing that the successor of each blue end node is the white start node of the circularly next employee. Subfigure 1b shows a sub-circuit solution, where employee 1 services (in order) scooters A, B, and C; employee 2 services scooters D and E; employee 3 services no scooters; and no employee services scooter F.

Since the `subcircuit` constraint allows some scooters not to be serviced (by their nodes having self-loops in the visualised solution) and since we need to decide, for each scooter, which employee services it, we introduce a dummy employee that services all the scooters serviced by no employee. We do this by introducing an employee called `dummy` and requiring that a scooter be serviced by that employee if and only if its node has a self-loop in the visualised solution.

A skeleton MiniZinc model and instances of varying sizes and difficulty, in the form of datafiles using the parameter names above, are [available](#). For brevity, you need not import lines 1 to 54 of the skeleton model, if you do not change them, into your report.

Tasks

Perform the following sequence of tasks:¹

- A. Write and evaluate a MiniZinc model called `SSP-A.mzn` in order to solve the SSP.

Hint: We provide in the skeleton model most of the decision variables that are needed to model this problem, but you can remove some variables if you find no use for them, and you might need additional variables.

¹Solo teams, except PhD students, may skip Task B, but are highly encouraged to do it nevertheless.

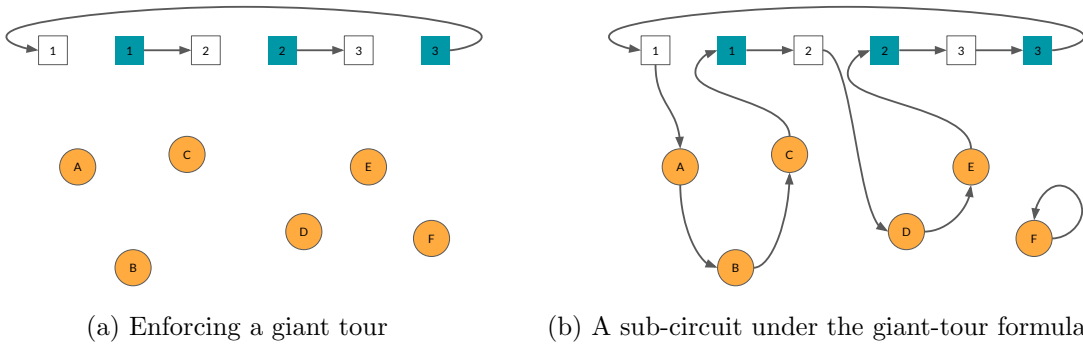


Figure 1: The giant-tour viewpoint

Warning: It is very rare but you may run into what is called *numerical instability* (due to integers being relaxed into floats) when a MIP-based backend claims that a feasible sub-optimal solution (as per the other backends) is optimal. Sometimes, very small changes to the model remove the bug, but do not spend much time on this and just mention the instability in reply to the question “Are there any contradictions between the results?” of the report.

- B. Extend `SSP-A.mzn` by adding the statement `include "gecode.mzn"` and adding the annotations `restart_constant(ρ)` and `relax_and_reconstruct(Succ, σ)` to the `solve` statement (just *before* the `minimize` keyword), for Gecode to use large-neighbourhood search (LNS) for solving the SSP. Call this new model `SSP-B.mzn`.

Determine experimentally (but without reporting how) good values for ρ (denoting the constant number of nodes after which the solver restarts) and σ (denoting the probability, as an integer percentage, of each variable of the array `Succ` being fixed upon a restart to its value in the previous solution): try a few pairs $\langle \rho, \sigma \rangle$ in $\{3000, 4000, \dots, 10000\} \times \{60, 70, 80, 90\}$, on the largest instances. Evaluate `SSP-B.mzn`, only under Gecode: when invoking the script `run_backends.sh` use the flag `--backends gecode`.

With LNS, Gecode no longer performs systematic search, but *local search*, and is thus no longer able to prove optimality unless the objective value reaches the lower bound, but it can often find solutions of much higher quality before timing out. Based on your evaluation, is this a reasonable trade-off here? Why?

For the evaluation(s), use all the provided instances. ***Use a time-out of 5 CPU minutes per instance in order to avoid too long solving times.*** For your convenience, here are the minimal objective values for some small instances (in real life, you do not know any of the optima when you start modelling a problem; do not ask the helpdesk whether the `-264` in the second row is wrong: that instance was crafted for catching a common modelling error):

instance name	objective value
005_003_01	-266
009_006_03	-264
010_008_01	-636
012_006_03	-538
023_017_03	-1239

Report Instructions

1. You ***must*** fill in the [provided skeleton report and skeleton model\(s\)](#). You ***must*** inspect your model(s) using our [checklist](#). See the [demo report \(L^AT_EX source\)](#) for generic instructions and for the expected quality of a report. The specific writing instructions are in comments that start with ‘%%’ in the L^AT_EX source of the provided skeleton report and should be followed even when not using L^AT_EX. All running text should be black: before submitting, comment away line 15 (which typesets the placeholders in [blue](#)) and uncomment line 16 (which typesets them in black). In the provided MiniZinc skeleton model(s), the placeholders are ‘...’ and the main model items are delimited by comments that start with ‘%%’.
2. Spellcheck the report and the comments in ***all*** models, in order to show both self-respect and respect for your readers. Thoroughly proofread the report, at least once per teammate, and ideally grammar-check it. In case you are curious about technical writing: see the [English Style Guide of UU](#), the technical-writing [Style Manual of the Optimisation research group](#), a list of [common errors in English usage](#), and a list of [common errors in English usage by native Swedish speakers](#).

3. Remember that when submitting you implicitly certify (a) that your report and all its uploaded attachments were produced solely by your team, except where explicitly stated otherwise and clearly referenced, (b) that each teammate can individually explain any part starting from the moment of submitting your report, and (c) that your report and attachments are not freely accessible on a public repository.
4. Submit (by only *one* of the teammates) by the given *hard* deadline two files via *Studium*: your report as a *PDF* file (all other formats will be rejected) and a compressed folder in *ZIP* format with all source-code files (including all MiniZinc models, plus possibly a software pipeline for the project) and datafiles (only for the project).

Grading Rules (stricter than the ones for Assignment 1)

If all tasks have been tackled, *and* all requested models are uploaded in files with the imposed names, structure, comments (in the model), and explanations (in the report) exemplified in the demo report, *and* all models ultimately produce *correct* solutions to *all* instances under backends of *all* considered solving technologies, *and* all models produce (*near-*)*optimal* solutions in *reasonable* time to *most* instances under backends of *at least two* technologies (under MiniZinc version 2.9.4 on a Linux computer of the IT department), *then* you get a score of at least 1 point (read on), *else* your *final* score is 0 points. Furthermore:

- *If* all models have *good enough* comments and explanations *and* all task answers are *correct enough*, *then* you get a *final* score of 3 or 4 or 5 points and are *not* invited to the grading session.
- *If* some models have *poor* comments or explanations *or* some task answers have *severe errors*, *then* you get an *initial* score of 1 or 2 points and *might* be invited to the grading session, at the end of which you are informed whether your initial score is increased or not by 1 point into your *final* score. A non-invitation leads to your final score being the initial one, and the same holds for each invited student who makes a no-show.

Also, *if* an assistant figures out a minor fix that is needed to make some model run as per our instructions above, *then*, instead of giving 0 points, the assistant may at their discretion deduct 1 point from the score earned upon the fix.