Modelling for Combinatorial Optimisation (1DL451) and Constraint Programming (1DL442) Uppsala University – Autumn 2023 Assignment 3: Scooter Service Problem (SSP)

Prepared by Gustav Björdal and Pierre Flener

— Deadline: 13:00 on Friday 6 October 2023 —

The scope of this assignment is Topics 1 to 8: you need ***not*** show any knowledge of subsequent topics. The source code of the demo report (at `https://user.it.uu.se/~pierref/courses/COCP/demoReport`) has problem-independent indications on how to proceed. Read the *Submission Instructions* and *Grading Rules* at the end of this document. 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
```

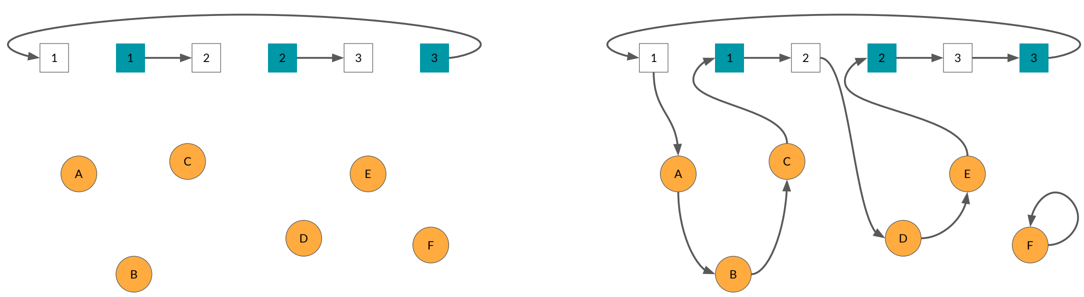
where `alpha`, `beta`, and `gamma` are additional parameters. Note that we do not minimise the total travel time of all the employees: this can be done as a post-processing step where we optimise the route of each employee, but we do not consider doing so in this assignment.

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) Enforcing a giant tour (b) A sub-circuit under the giant-tour formulation

Figure 1: The giant-tour viewpoint

A skeleton MiniZinc model and instances of varying sizes and difficulty, in the form of datafiles using the parameter names above, are at `https://user.it.uu.se/~pierref/courses/COCP/assignments/assignment3/assignment3.zip`. 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. You may use a search annotation towards greatly accelerating CP and LCG backends: first state a suitable search strategy in plain English and argue for it, and then formulate or approximate it as a MiniZinc search annotation.

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 report the instability in reply to the question "Are there any contradictions between the results?" of the demo 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 (an integer from 0 to 10000 denoting the constant number of nodes after which a restart will be made) and σ (an integer from 0 to 100 denoting the probability, as a percentage, of each variable of the array `Succ` being fixed upon a restart to its value in the previous solution) and evaluate `SSP-B.mzn` (in the usual way), but only under the Gecode solver: when invoking the script `run_backends.sh` use also the flag `--backends gecode`. Some other solvers also support LNS, but currently not via their MiniZinc backends.

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 all the optima when you start modelling a problem; note that −264 in the second row is *not* wrong):

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

[1]Solo teams may skip Task B, but are highly encouraged to do it nevertheless.

Submission Instructions

In order to protect yourself against an unnecessary loss of points, use the following to-do list before submitting:

- Tackle ***each*** task of ***each*** problem, using (in order to accelerate the grading) the numbering and the ordering in which they appear in this assignment statement.
- Take the instructions of the demo report at `https://user.it.uu.se/~pierref/courses/COCP/demoReport` as a ***strict*** guideline for the structure and content of a model description, model evaluation, and task answer, and as an indication of the expected quality of content: write with the precision that you would expect from a textbook.
- You ***must*** use the MiniZinc experiment script explained in `https://user.it.uu.se/~pierref/courses/COCP/assignments/cheatsheet.pdf`: it conducts the experiments and generates a result table that can be automatically imported (rather than manually copied) into a LaTeX report, so that each time you change a model, it suffices to re-run the script and re-compile your report, without any tedious number copying!
- If a MiniZinc model does not compile and run error-free under backends of ***all*** the considered solving technologies, then obtain a teacher's approval in due time ***before*** submitting your report.
- ***Thoroughly*** proofread, spellcheck, and grammar-check the report, at least once per teammate, including the comments in ***all*** code. In case you are curious about technical writing: the *English Style Guide* of UU at `https://mp.uu.se/en/web/info/stod/kommunikation-riktlinjer/sprak/eng-skrivregler` and the technical-writing *Checklist & Style Manual* of the Optimisation group at `https://optimisation.research.it.uu.se/checkList.pdf` offer many pieces of advice; common errors in English usage are discussed at `https://brians.wsu.edu/common-errors`; in particular, common errors in English usage by native Swedish speakers are listed at `https://www.crisluengo.net/english-language`.
- Match ***exactly*** the uppercase, lowercase, and layout conventions of any filenames and I/O texts imposed by the tasks, as we will process submitted source code automatically. However, do not worry when *Studium* appends a version number to the filenames when you make multiple submission attempts until the deadline.
- Do ***not*** rename any provided problem-specific skeleton model, for the same reason.
- Import all the MiniZinc models ***also*** into the report: for brevity, it is allowed to import only the lines after the copyright notice.
- Produce the report as a ***single*** file in ***PDF*** format; all other formats will be rejected.
- 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.
- Submit (by only ***one*** of the teammates) the solution files (one report and all MiniZinc source code) ***without*** folder structure and ***without*** compression via *Studium*, whose clock may differ from yours, by the given ***hard*** deadline.

Grading Rules

If *all* the tasks have been tackled, ***and*** *all* the requested models are in files with the imposed names, comments, and explanations exemplified in the demo report, ***and***, for *all* our grading instances, *all* models produce *correct* outputs under backends of *all* the considered solving technologies and (*near-*)*optimal* outputs under backends of *at least two* technologies in *reasonable* time under MiniZinc version 2.7.6 on a Linux computer of the IT department, ***then*** you get at least 1 point (read on), ***else*** your *final* score is 0 points. Furthermore:

- ***If*** *all* models are good (in terms of comments and against the checklists at `https://user.it.uu.se/~pierref/courses/COCP/demoReport/checklist.pdf`) ***and*** *all* task answers are *mostly correct*, ***then*** you get a *final* score of 3 or 4 or 5 points and are *not* invited to the grading session. Note that we have not been able to design the equivalent of Table 1 of Assignment 2, where some objective criteria reliably assess the quality of a model, so grading is more subjective here: this is further preparation for your project and real life, where one often does not know all the optima either.

- ***If*** *some* models are flawed (in terms of comments and against the checklists of Topics 2 and 3) ***or*** the task answers have *many errors*, ***then*** you get an *initial* score of 1 or 2 or 3 points and *might* be invited to the grading session, where you can try and increase your initial score by 1 point into your *final* score, a no-show or non-invitation leading to your final score being the initial one.

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