Algorithms and Data Structures 3 (course 1DL481)
Uppsala University – Spring 2025
Assignment 1

Prepared by Pierre Flener, Di Yuan, and Gustav Björdal

— Deadline: 13:00 on Friday 7 February 2025—

The assignments exercise the lecture material that is ***not*** covered by the exam. You ***must*** fill in the skeleton report in the provided materials, by following its instructions. Read the **Submission Instructions** and **Grading Rules** at the end of this document ***before*** tackling the following problems upon reading them a first time. Note that a problem *weight* indicated below might ***not*** reflect the relative *time* effort (within the assignment) that you actually spend on that problem: some points might be easier to earn. It is strongly recommended to prepare and attend the help sessions, as huge time savings may ensue.

Problem 1: Mixed Integer Programming (MIP) (40% weight)

The aim of solving this problem is to get hands-on experience with declaratively encoding an important real-world problem into a linear objective function and a system of linear constraints over integer and real-number variables, towards impressive performance.

Background and Motivation: The following optimisation problem is motivated by locating emergency-response *service stations*, which have fire trucks, for example. In the planning process, the *service region* is divided into zones. Each *zone* represents a candidate location of an emergency-response service station. Each zone has a *demand*: the expected frequency of service request, based on historical data and estimation. When the emergency-response agency receives a call, originating from some zone, the operator usually dispatches a vehicle from the closest (measured in travel time) service station, which may or may not be in the same zone.

For a given planning solution, the service quality for a zone is based on the travel time of the closest vehicles. To account for the fact that one or even multiple vehicles of the closest service station may become occupied for serving other requests, we consider the average travel time of the closest vehicles, which may be located at different service stations. Furthermore, this average time is multiplied with the demand, so zones with high demand will receive more weight in the optimisation process.

Problem Specification: The *service station location problem* has the following inputs:

- z is the number of zones in the service region;
- s is the number of service stations to be located within the zones;
- v is the number of vehicles hosted at each service station;
- c is the number of vehicles considered for the average travel time of the closest vehicles;

- $Demand[i]$ is the demand of zone i, with $i \in 1..z$;
- $Time[i,j]$ is the travel time from zone i to zone j; note that the matrix need not be symmetric (hence pay attention to get the indices right); the travel time within a zone is typically a small value, though not zero.

The optimisation decision consists in selecting exactly s out of the z zones in order to locate service stations in distinct zones. The objective is to minimise the sum over all the zones of the average travel time of the c closest vehicles of each zone, multiplied by the demand of the zone.

Data Format: The problem parameters in the AMPL data files in the provided folder `servStatLoc-data` are below, for $z = 10$ zones, $s = 2$ service stations, $v = 2$ vehicles per service station, and $c = 3$ closest vehicles:

```
param z := 10;
param s :=  2;
param v :=  2;
param c :=  3;
param Demand :=
  1   0.000803429155663251
  2   0.000951406002100867
   ...
 10   0.000597429377117032;
param Time :=
  1   1  0.0010
  1   2  1.6420
   ...
  1  10  4.0630
  2   1  1.6420
   ...
 10  10  0.0010;
```

where:

- the parameter `Demand`, indexed over the zones, gives the z demand values $Demand[i]$;
- the parameter `Time` has z^2 elements specifying the travel times $Time[i,j]$;
  for example, the travel time from zone $i = 1$ to zone $j = 10$ is 4.063 in the data above.

In the provided AMPL model `servStatLoc.mod`, the following declarations match up with the provided AMPL data files:

```
param z; # number of zones
param s; # number of service stations
param v; # number of vehicles per service station
param c; # number of closest vehicles for average
set Zones := 1..z;
param Demand {Zones} >= 0; # Demand[i] = demand of zone i
param Time {Zones,Zones} >= 0; # Time[i,j] = time from zone i to j
```

See the AD3 Resources for free-of-charge access to the world-class commercial MIP solver Gurobi via AMPL.

Tasks: You ***must*** fill in the placeholders in the skeleton report and skeleton model in the provided materials, by following their instructions:[1]

A. Describe a model of the service station location problem as a mixed integer linear program: what are the variables, their meanings, their constraints, and the objective function? Do not worry about symmetry in the search space: MIP solvers automatically exploit symmetries. **Hint:** For each zone, the actually closest vehicles can be determined by the minimisation process without representing them in the model, so that it suffices to represent the number of vehicles of each zone that serve each zone.

B. Implement the resulting model into `servStatLoc.mod` and upload it without explaining and including it in the report, but indicate there the MIP solver and hardware that you used for your experiments in the next tasks. All the syntax you need is in our MIP slides. Four points will be deducted from your score if your model has a nonlinear objective function or nonlinear constraints, such as the quadratic constraints and logic constraints that language allows: check if the AMPL output mentions non-linearity. One (possibly additional) point will be deducted from your score if your uploaded model does not correspond to the model you describe for Task A.

C. Solve the problem for $z = 10, s \in 2\,..\,4, v = 2, c = 3$ using the data files `location-010-0`s`-2` in order to see the impact on the optimal objective value when parameter `s` increases, as one then has a higher budget for service stations and more vehicles overall. For your convenience: the optimal objective value for $s = 2$ is 0.008740338682.

D. Solve the problem for $z = 20, s \in 2\,..\,6, v = 2, c = 3$ using the data files `location-020-0`s`-2` in order to see the impact on the optimal objective value when parameter `s` grows beyond 4. For your convenience: the optimal objective value for $s = 2$ is 0.02324626135.

E. Solve the problem for $z = 40, s = 5, v = 2, c = 3$ using the data file `location-040-05-2`.

F. Solve the problem for $z = 80$ and $c = 3$ using the data files `location-080-08-2` and `location-080-16-1` in order to see the impact on the optimal objective value when distributing 16 vehicles for $s = 8$ with $v = 2$ versus $s = 16$ with $v = 1$.

G. Solve the problem for $z = 120, s = 10, v = 2, c = 3$ using the data file `location-120-10-2`. If your model times out, then propose an algorithm for delivering a not necessarily optimal solution, such that the algorithm is expected to take reasonable running time: describe the steps of the algorithm, but you do not need to implement and run it.

H. Solve the problem for $z = 250, s = 12, v = 3, c = 4$ using the data file `location-250-12-3`. If your model times out (but not on Task G), then follow the timeout instructions of Task G.

I. Express and justify the size of the search space of a totally brute-force search algorithm in terms of the problem parameters z, s, v, and c. For each instance mentioned above that your used MIP solver solved to proven optimality without timing out, state how many candidate solutions this brute-force search algorithm has to examine per second in order to match your reported runtime performance.

[1] Solo teams (except PhD students) may omit the potential algorithm design parts of Task G or H (or both), as well as the entire Task I, but they are encouraged to perform them nevertheless.

Use a timeout of at least 300 seconds per run and report the performance (runtime, objective value, and optimality gap) over a ***single*** run per instance, as the recommended solvers are deterministic by default; a precision of two decimal places suffices here for the runtime and optimality gap, but the objective value should be given in full precision. With Gurobi, use the AMPL command `option solver gurobi; option gurobi_options 'outlev=1';` before you run `solve` in order to turn on verbose printing, which includes the optimality gap.

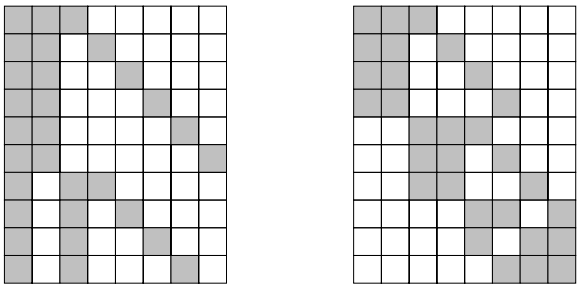
Reminder. MIP solvers are ***exact***: for a minimisation problem, the *optimality gap* is the relative difference between the current upper bound u (the objective value of the currently best solution) and current lower bound ℓ (the objective value of the currently best leaf node) on the objective function, that is the ratio $\frac{u-\ell}{\ell}$, when a MIP solver is stopped prematurely; if $\ell = u$ then the optimality gap is zero and the MIP solver has actually proved the optimality of its currently best solution.

Problem 2: Stochastic Local Search (SLS) (60% weight)

The aim of solving this problem is to get hands-on experience with procedurally encoding a hard combinatorial optimisation problem into an SLS algorithm that can near-optimally if not optimally solve the problem, with potentially impressive performance.

The *investment design problem* is about finding a matrix of v rows and b columns of 0-1 integer values, such that each row sums up to r, with $v \geq 2$ and $b \geq r \geq 1$, and the largest dot product between all pairs of rows is minimised. Equivalently, one has to find v subsets of size r within a given set of b elements, such that the largest intersection of any two of the v sets has minimal size. An instance of the problem is parametrised by a triple $\langle v, b, r \rangle$.

For example, the following figure shows two $\langle 10, 8, 3 \rangle$ investment designs, where grey cells represent value 1 and white cells represent value 0:



In these two investment designs, there are dot products (or: intersection sizes) of 0 to 2, so their largest dot products are both 2: this is minimal, as there exists no $\langle 10, 8, 3 \rangle$ investment design where 1 is the largest dot product [1].

This is an abstract description of a problem that appears in finance (see The Big Short), where the rows are the baskets (also known as subpools or tranches) of an investment portfolio and the columns correspond to the credits the baskets can invest in, so that the baskets are of equal size but minimal overlap. In a typical investment design in finance, we have $4 \leq v \leq 25$ and $250 \leq b \leq 500$, with $r \approx 100$.

A lower bound on the number λ of shared elements of any pair among v subsets of size r drawn from a given set of b elements is given in [1]:

$$\mathrm{lb}(\lambda) = \left\lceil \frac{\left\lceil \frac{rv}{b} \right\rceil^2 ((rv) \bmod b) + \left\lfloor \frac{rv}{b} \right\rfloor^2 (b - ((rv) \bmod b)) - rv}{v(v-1)} \right\rceil \tag{1}$$

For 6 of the following 7 instances, the lower bound on λ is known to be feasible:

v	b	r	$\mathrm{lb}(\lambda)$
10	360	120	32
15	350	100	24
11	150	50	14
10	100	30	7
19	19	9	4
15	21	7	2
12	44	11	2

For the instance $\langle 15, 21, 7\rangle$ in the table, the lower bound 2 is infeasible and a best solution has $\lambda = 3$, but you are ***not*** allowed to exploit that knowledge in your algorithm. For the instance $\langle 10, 8, 3\rangle$ discussed above, the lower bound is 1, which is infeasible [1].

Tasks: You ***must*** fill in the placeholders in the skeleton report in the provided materials, by following its instructions. Note that performing Tasks A to C is best done in parallel:[2]

A. **SLS Algorithm.** Design an SLS algorithm for solving the investment design problem, by performing the following problem-independent steps while reading the skeleton code of Task B and addressing each of them in the report. Start with a very simple algorithm and test it on small instances; improve it only if it is too slow and test it on larger instances, etc, until you are satisfied with the performance (see the end of Task C for our requirements):

   1. Representation. Describe how to represent the problem: what are the variables, their meanings, their constraints, and the objective function?
   2. Initial Assignment. Describe an algorithm for generating (fast) a randomised initial assignment. **Hint**: Satisfy some or all of the constraints in the initial assignment.
   3. Move. Describe one or more *moves* that go from an assignment to a neighbouring assignment by changing the values of a few variables. **Hint**: If you followed the previous hint, then preserve the satisfaction of the chosen constraints by every move: your SLS algorithm will then always be at an assignment satisfying all those constraints and can focus on trying to satisfy the remaining constraints and optimising the *objective value* (that is, the value of the objective function).
   4. Constraints. Describe for each constraint how its satisfaction is either algorithmically checkable efficiently or guaranteed to be preserved by the previous two design choices.
   5. Neighbourhood. Describe a neighbourhood based on the proposed moves. Derive a formula for computing the size of the neighbourhood in terms of the problem parameters. Discuss whether the neighbourhood makes the search space *connected*, in the sense that every *feasible assignment* (that is, every assignment satisfying all the constraints, whether optimal or not) is reachable from every initial assignment (you only need to sketch a proof if the search space is connected, and give a counterexample otherwise). Note that the neighbourhood should *not* be of size 1: otherwise you are describing a greedy algorithm. **Hint**: Do not over-engineer the neighbourhood.
   6. Cost Function. Describe a cost function, whose value is to be minimised during search. Note that the cost function need not be equal to the objective function, but the cost value should decrease when the objective value decreases. Also note

[2]Solo teams (except PhD students) only need to evaluate *one* configuration of values for the hyperparameters in Task C and they may omit Task D, but they are encouraged to perform it nevertheless.

that one can switch during search between alternative cost functions. **Hint**: If two assignments have the same objective value but one of them is somehow better, then a good cost function gives a lower value to the better assignment. For example, imagine the two equally good $\langle 10, 8, 3\rangle$ investment designs in the drawings at page 4 were not optimal: the cost function can be based on a tiebreaker expressing why the right design is closer to a better design than the left one.

7. Probing. Describe how a neighbouring assignment, as reachable by a move, can be *probed* efficiently: describe how the cost function can be evaluated efficiently and incrementally, and describe the data structures used to do so. Give, without proof, the time complexity of probing; ideally, it is (sub-)linear in the problem parameters.
8. Heuristic. Describe a heuristic for *exploring* (via probing) your neighbourhood of Step 5 and *selecting* a neighbouring assignment to *commit* to. State whether the neighbourhood is explored exhaustively and, if so, how you determine when it was exhausted. Explain how you ensure that the same neighbour is not probed twice during a given exploration. **Hint**: It can be beneficial to explore the neighbourhood in a random order.
9. Optimality. Describe how you use a bound on the objective value in order to terminate sometimes the search with proven optimality, as part of the heuristic.
10. Meta-Heuristic. Describe a meta-heuristic based on tabu search: explain how the tabu list is represented; choose (a formula for) its size; explain how fine-grained its content is; and describe how it can be looked up and maintained efficiently. Note that the tabu list is not necessarily an actual list, but rather a concept. Make sure that worsening moves are sometimes made.
11. Random Restarts. Describe how to detect or guess that a random restart should be made, as part of the meta-heuristic.
12. Optional Tweaks. Describe ideas that you used in order to improve your algorithm.

Most of your effort should be spent on steps 6, 7, and 10. Identify the hyperparameters in each step.

B. **Implementation.** Implement — in either Java, using the skeleton code in the provided folder `invDes`, or any other ***high-performance*** programming language for which a compiler or interpreter is available on the Linux computers of the IT department (see the AD3 Resources) — the SLS algorithm designed in Task A and upload it without including it in the report, but give there its compilation and running instructions. Because speed is of the essence for SLS, we recommend against Python for example.

An executable called `InvDes` must read the problem parameters v, b, r as command-line arguments, say `./InvDes 10 8 3` or `java InvDes 10 8 3`, and write to standard output a line with the space-separated values of v, b, r, the lower bound $\mathrm{lb}(\lambda)$ on λ, and the achieved λ, followed by one line per row of a $v \times b$ matrix representing the solution, the 0-1 cell values being space-separated. For example, the $\langle 10, 8, 3\rangle$ investment design on the left at page 4 is represented by the provided file `invDes-10-8-3.txt`.

You ***must*** at least experimentally pipe the `InvDes` output into the provided polynomial-time solution checker, which is a Python script that reads a solution from standard input, in order to gain confidence in the correctness of your implementation, for example by `./InvDes 10 8 3 | ./invDesChecker`.

C. **Experiments.** Indicate the hardware that you used for your experiments. Determine (without reporting how) two good configurations of values for your identified hyperparameters and evaluate these configurations experimentally, using a table. Use a timeout of at least 300 seconds per run and report the ***median*** (not: average) performance (runtime, steps, and achieved λ) over ***at least*** 5 independent runs per instance and configuration, because an SLS algorithm ***must*** be randomised by definition; a precision of one decimal place suffices here.

Hint: In order to save a lot of time, it is very important that you write a script that conducts the experiments for you and directly generates a result table that is imported into your report (see the source code of the provided skeleton report for how to do that): each time you change your implementation, it suffices to re-run that script and re-compile your report, without any tedious number copying! The sharing of such a script is allowed and even encouraged. It is recommended to define an optional command-line flag, say `-p`, for the executable `InvDes` so as also to output the runtime (in seconds) and the number of steps in the first line of the output and to suppress the output of the $v \times b$ matrix representing the solution.

The larger instances in the table above may be difficult for your SLS algorithm, but you are expected to get quite close to their optimal values of λ mentioned above. A minimum requirement for score 5 is to be within one unit of the actual optimum.

D. **Exact Algorithm.** Outline (in plain English or in high-level pseudocode, but without an implementation) an exact algorithm (performing brute-force search, if you want) for the investment design problem. Express the size of the search space of your exact algorithm in terms of the problem parameters v, b, and r. For each instance in the table above that your SLS algorithm solves to proven optimality before timing out, state how many candidate solutions your exact algorithm would have to examine per second in order to match the runtime performance of the seemingly best configuration of values for the hyperparameters, according to Task C. Comment on those numbers.

References

[1] O. Sivertsson, P. Flener, and J. Pearson. A bound on the overlap of same-sized sets. *Annals of Combinatorics*, 12(3):347–352, October 2008. Available at https://dx.doi.org/10.1007/s00026-008-0355-0.

Submission Instructions

1. The report instructions are in comments that start with '%%' in the LaTeX source of the skeleton report and should be followed even when not using LaTeX. All running text should be black: comment away line 19 (which typesets the placeholders in blue) and uncomment line 20 (which typesets them in black).

2. Spellcheck the report and the comments in ***all*** code, 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 Check List & 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. 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. Do ***not*** rename any of the provided skeleton codes, for the same reason. However, do not worry when *Studium* appends a version number to the filenames when you make multiple submission attempts until the deadline.

4. 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.

5. Submit (by only ***one*** of the teammates) by the given ***hard*** deadline three files via *Studium*: your report as a ***PDF*** file (all other formats will be rejected) and two compressed folders in ***ZIP*** format with all source code required to run your experiments of each problem.

Grading Rules

For each problem:
If the requested source code is uploaded, ***and*** it runs *without* compilation or runtime errors on a Linux computer of the IT department under the compiler, interpreter, or solver that you indicate, ***and*** it computes *correct* and (*near-*)*optimal* solutions in *reasonable* time to *some* of our tests on that hardware, ***then*** you get a score of at least 1 point (read on), ***else*** your *final* score is 0 points. Furthermore:

- ***If*** the code *passes most* of our tests ***and*** the report addresses *all* tasks and subtasks in a *good enough* way, ***then*** you get a *final* score of 3 or 4 or 5 points and are *not* invited to the grading session for this problem.

- ***If*** the code *fails too many* of our tests, ***or*** the report does *not* address all tasks and subtasks, ***or*** *some* task or subtask answers have *severe errors*, ***then*** you get an *initial* score of 1 or 2 points and *might* be invited to the grading session for this problem, at the end of which you 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 your code 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.

Let s_i be your final score on Problem i, whose stated weight is w_i. Your final score on the entire assignment is $\sum_i (w_i \cdot 2 \cdot s_i)$, rounded to the ***nearest*** (and hence possibly previous) integer, provided $s_i > 0$ for both problems.

Considering there are three help sessions for each assignment, you ***must*** earn at least 3 points (of 10) on each assignment until the end of its grading session, including at least 1 point (of 5) on each problem and at least 10 points (of 20) over both assignments, in order to pass the *Assignments* part (2 credits) of the course, if you attend the guest lecture from industry.