Algorithms and Data Structures III (course 1DL481) Uppsala University – Spring 2023 Report for Assignment 1 by Team 7

Anderson LEONG Louis HENKEL

3rd February 2023

Problem 1: Mixed Integer Programming (MIP)

Task a: Model. We define the variables first:

- Let z be the number of zones.
- Let s be the number of service stations that we place in one zone each.
- Let v be the number of vehicle, we will therefore have $N = v \cdot s$ of vehicles at our disposal in total.
- Let c be the number of vehicles considered
- Let d be a tuple of length z giving the demand for each zone z.
- Let T be a matrix of dimension $z \times z$ with the travel time from one zone to the other.

We create a cost matrix C, which is generated from multiplying each line of the time matrix T_i with the i-th value of the demand tuple. In addition we have a binary vector $\mathbf{s}$ of length z where the i-th value is 1 if we have a station zone i. Finally we have a matrix A of dimension $z \times z$. We have these three constrains:

$$A_{i,j} \in [0, v] \quad \forall i, j \in [1, z] \tag{1}$$

$$\sum_{i=1}^{z} A_{i,j} = c \quad \forall j \in [1, z] \tag{2}$$

$$\|\mathbf{s}\| = s \tag{3}$$

Our cost function is given by:

$$\mathsf{cost} = \sum_{i=1}^{z} \sum j = 1^z A_{i,j} \cdot C_{i,j}$$

Task b: Implementation. Our model `servStatLoc.mod` is uploaded with this report: we checked that its constraints and objective function are linear (and we are aware that four points will otherwise be deducted from our score for this problem). We chose the MIP solver Gurobi for our experiments, which we ran on the NEOS server for MINTO/AMPL with a CPU time set to 4 hours.

Task c: 10 Zones. The results are in Table 1. When s increases, the optimal objective value decrease, because we have more stations, therefore the average distance is lower, because the new stations may be closer to some of the zones then the initial stations.

Task d: 20 Zones. The results are in Table 1. When s grows beyond 4, the optimal objective value is 0.011920 for $s = 5$.

Task e: 40 Zones. The results are in Table 1.

Task f: 80 Zones. The results are in Table 1. Upon $s = 16$ service stations with $v = 1$ vehicle each, the optimal objective value is 0.013908 the one for $s = 8$ and $v = 2$, because 0.116972.

Task g: 120 Zones. The results are in Table 1. Our model does not time out.

Task h: 250 Zones. The results are in Table 1. Our model does not time out.

Task i: Brute-Force Algorithm. The size of the search space of a totally brute-force search algorithm is

$$\binom{z}{s} = \frac{z!}{s! \cdot (z-s)!} = \frac{z \cdot (z-1) \cdots (z-s+1)}{1 \cdot 2 \cdots s},$$

because if we want to position s objects in z places. The first object we position has z places it can go to, the second one hase $z-1$ possibilities, and so on until the last one that $z-s+1$ options. As the order in which we make position the station does not matter, we have to divide by the possibilities in which we can position the station. High school basic combinatorics therefore give us the result above.

The numbers of candidate solutions this brute-force search algorithm has to examine per second in order to match the reported runtime performance of Gurobi on our model are given in the right-most column of Table 1, for each instance that Gurobi solved to proven optimality without timing out.

z	s	v	c	time	objective value	optimality gap	brute-force
10	2	2	3	0.00	0.008740	0.00%	∞
10	3	2	3	0.01	0.007145	0.00%	$1,2 \cdot 10^4$
10	4	2	3	0.01	0.005884	0.00%	$2,1 \cdot 10^4$
20	2	2	3	0.02	0.023246	0.00%	$9.5 \cdot 10^3$
20	3	2	3	0.02	0.016681	0.00%	$5.7 \cdot 10^4$
20	4	2	3	0.02	0.013908	0.00%	$2.423 \cdot 10^5$
20	5	2	3	0.03	0.011920	0.00%	$5.168 \cdot 10^5$
20	6	2	3	0.01	0.010839	0.00%	$3.876 \cdot 10^6$
40	5	2	3	0.34	0.048470	0.00%	$1.935 \cdot 10^6$
80	8	2	3	8.06	0.124612	0.00%	$3.596 \cdot 10^9$
80	16	1	3	0.06	0.116972	0.00%	$4.493 \cdot 10^{17}$
120	10	2	3	35.94	0.231958	0.00%	$4.848 \cdot 10^{12}$
250	12	3	4	383.36	0.523060	0.00%	$2.482 \cdot 10^{17}$

Table 1: Service station location: runtime (in seconds), objective value, and optimality gap (in percent; positive if an optimal solution was not found and proven before timing out) using Gurobi, with a timeout of 14400 CPU seconds. The right-most column gives the number of candidate solutions the brute-force search algorithm has to examine per second in order to match the runtime performance of Gurobi, if the instance was solved to proven optimality, and 'n/a' for 'non-applicable' otherwise.

Problem 2: Stochastic Local Search (SLS)

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 [?]:

$$\mathrm{lb}(\lambda) = \left\lceil \frac{\left\lceil \frac{rv}{b} \right\rceil^2 ((rv) \bmod b) + \left\lfloor \frac{rv}{b} \right\rfloor^2 (b - ((rv) \bmod b)) - rv}{v(v-1)} \right\rceil \tag{4}$$

Task a: SLS Algorithm.

1. Representation. The current state of the local search is given by a $v \times b$ matrix, where each row v is a binary array of length b where there are always r elements that are 1.

2. Initial Assignment. It is made by randomly generating v binary arrays where r ones are randomly positioned.

3. Move. We randomly select one row, and check which one the fields with a one is the most expensive and which field with a zero is the cheapest. If they reduce the costs, then we make the switch, else we move to the next row. With a probability of α we instead make a random move, and with a probability β we make a random restart.

4. Constraints. Each row must have exactly r elements which are one, which is a built in constrain which is never violated, because we only make swaps in the moves and each row gets initialised this way.

5. Neighbourhood. As for each of the v rows, we create one possible move, there is a total of v moves in the neighbourhood. The search space in connected, as each permutation of r from b elements can be found from a finite sequence of permutations and switches. As we can reach every feasible configuration for 1 row, we can also reach every feasible configuration of the matrix.

6. Cost Function. Let v_i and v_j be the i-th and j-th row of a configuration. Then our cost function is given by:

$$\mathsf{cost} = \max_{i \neq j} v_i \cdot v_j$$

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) the neighbourhood and selecting a neighbouring assignment to commit to. Decide if the neighbourhood is to be explored exhaustively and, if so, how you determine when it was exhausted. Explain how to ensure that the same neighbour is not probed twice during a given exploration.

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 either tabu search (strongly recommended for the investment design problem) or simulated annealing. In case of tabu search: explain how the tabu list is represented; choose (a formula for) its size; explain how fine-grained or coarse-grained its content is, conceptually; 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. In case of simulated annealing: explain how you chose the cooling function, cooling rate, and initial temperature.

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 improvements such as aspiration, diversification, or other ideas that you use to improve your SLS algorithm.

In summary, the local-search parameters (not the problem parameters v, b, r) are α and β.

Task b: Implementation. We chose the high-performance programming language C++, for which a compiler is available on the Linux computers of the IT department. All source code is uploaded with this report. The compilation and running instructions are $\langle \dots \rangle$.

An executable called `InvDes` reads the problem parameters v, b, r as command-line arguments and writes to standard output a line with the space-separated values of v, b, r, the lower bound $\text{lb}(\lambda)$ 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.

We validated the correctness of our implementation by checking its outputs on many instances via the provided polynomial-time solution checker.

Task c: Experiments. All experiments were run under Linux Ubuntu 18.04 (64 bit) on an Intel Xeon E5520 of 2.27 GHz, with 4 processors of 4 cores each, with a 70 GB RAM and an 8 MB L3 cache (a ThinLinc computer of the IT department).

We fine-tuned the local-search parameters as follows. Discuss the impact of the local-search parameters α and β on the performance of your SLS algorithm.

The median runtime (in seconds), median number of steps, and median achieved λ over 5 independent runs for each of the 21 instances of the assignment instructions are given in Table 2, for two configurations of values for the local-search parameters α and β. The timeout was 300.0 CPU seconds per run.[1]

We observe that $\langle \dots \rangle$, because $\langle \dots \rangle$.

Task d: Exact Algorithm. An exact algorithm could work as follows: It could perform brute force search, trying out all possible non-equivalent configurations. Two configurations are equivalent, if a permutations of the rows leads to both configurations to be the same. This would lower the size of the search space. For one row, the search space is given by $\binom{v}{r}$. As we have a total of b rows, there are in total $\binom{v}{r}^b$ possibilities. As the permutation of b elements yield a group of size $b!$, the number of equivalent classes is $b!$. Hence, the total search space is of size

$$\frac{\binom{v}{r}^b}{b!}$$

[1]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 (see the LaTeX source code of Table 2 for how to do that), which is then automatically imported, rather than manually copied, into your report: each time you change the code, it suffices to re-run that script and re-compile your report, without any tedious copying! The sharing of scripts is allowed and even encouraged.

				$\langle\alpha,\beta\rangle = \langle 10,5\rangle$			$\langle\alpha,\beta\rangle = \langle 20,8\rangle$			
v	b	r	lb(λ)	time	steps	λ	time	steps	λ	exact
10	30	9	2	300.0	123	3	45.0	678	2	10^{22}
12	44	11	2	300.0	901	3	234.5	5678	2	10^{21}
15	21	7	2	300.0	1023	4	300.0	6789	3	n/a
16	16	6	2	123.4	567	2	89.1	2345	2	10^{14}
9	36	12	3							
11	22	10	4							
19	19	9	4							
10	37	14	5							
8	28	14	6							
10	100	30	7							
6	50	25	10							
6	60	30	12							
11	150	50	14							
9	70	35	16							
10	350	100	22							
13	250	80	22							
10	325	100	24							
15	350	100	24							
9	300	100	25							
12	200	75	25							
10	360	120	32							

Table 2: Investment design: median runtime (in seconds), median number of steps, and median achieved λ, for two configurations of values for the local-search parameters α and β, over 5 independent runs per instance, with a timeout of 300.0 CPU seconds per run. The right-most column gives the number of candidate solutions the outlined exact algorithm has to examine per second in order to match the runtime performance of the seemingly best configuration of values for the local-search parameters, namely $\langle\alpha,\beta\rangle = \langle 20,8\rangle$, if the instance was solved to proven optimality, and 'n/a' for 'non-applicable' otherwise. (The sample performance of this demo report is made up!)

The number of candidate solutions this exact algorithm has to examine per second in order to match the runtime performance of the seemingly best configuration of values for the local-search parameters, according to Task c, of our stochastic local search algorithm is given in the right-most column of Table 2, for each instance solved to proven optimality. We think that $\langle\ldots\rangle$, because $\langle\ldots\rangle$.

Feedback to the Teachers