University of Cape Town

STA5071

Simulation and Optimisation

Replicating a SA Approach to Solving the GMS Problem

Author:
Raisa Salie

Student Number:
SLXRAI001

January 4, 2022

Contents

1 Introduction

The general maintenance scheduling (GMS) problem aims to find an optimal planned preventative maintenance schedule for generating units in a power system under certain constraints (for example, power demand and manpower availability). The problem is combinatorial in nature, and is typically too complex for exact solution methods such as non-linear programming. Simulated annealing as a tool frequently used to solve this problem , and there are a number of conventions for doing so [2].

This report aims to reproduce the algorithms employed by Schlunz [2], in particular those that produced the best results in the paper for the 32- and 21-unit test systems run over 52 weeks. The best results were obtained using a particular configuration of settings which we will attempt to replicate in our version.

2 Mathematical Formulation of the Problem

2.1 Definitions of Mathematical Terms

The following terms are relevant to the formulation of our problem.

$$
\begin{aligned}
\mathcal{I} &= \text{set of indices of units } i \\
&= \{1, .., n\} \\
\mathcal{J} &= \text{set of indices of time periods } j \\
&= \{1, ..., m\} \\
x_{ij} &= \begin{cases} 1 & \text{if maintenance on unit } i \text{ commences at time } j \\ 0 & \text{otherwise} \end{cases} \\
\implies x_i &= j \\
y_{ij} &= \begin{cases} 1 & \text{if unit } i \text{ is under maintenance in period } j \\ 0 & \text{otherwise} \end{cases} \\
e_i &= \text{earliest allowed commencement time } j \text{ for unit } i \\
\ell_i &= \text{latest allowed commencement time } j \text{ for unit } i \\
d_i &= \text{duration of maintenance period for unit } i \\
g_{ij} &= \text{generating capacity of unit } i \text{ during period } j \\
D_j &= \text{load demand during period } j \\
S &= \text{safety margin as a proportion of } D_j \\
r_j &= \text{reserve level for time period } j \text{ (unused power)} \\
m'_{pij} &= \text{manpower required for unit } i \text{ if maintenance commences in period j} \\
M_j &= \text{maximum allowed manpower during period } j \\
\mathcal{K} &= \text{set of all indices of exclusion subsets} \\
&= \{1, 2, ..., K\} \\
K &= \text{the number of exclusion subsets} \\
I_k \subseteq \mathcal{I} &= k^{th} \text{ subset of } \mathcal{K} \\
K_k &= \text{maximum units within } I_k \text{ allowed for simultaneous maintenance} \\
W_{ext} &= maintenancewindowextensionparameter
\end{aligned}
$$

2.2 MIQP Formulation

Although the algorithm we are replicating follows a simulated annealing approach, there were some constraints that needed to be considered, which were described by the author in earlier problem formulations. The author formulated the problem as a mixed integer quadratic problem, for example. This formulation is described below.

$$\text{min.} \quad Z = \sum_{j=1}^{m} \left(D_j S + r_j\right)^2$$

s.t

$$\sum_{j=e_i}^{\ell_i} x_{ij} = 1, \qquad \forall i \in \mathcal{I} \tag{1}$$

$$x_{ij} = 0, \qquad j < e_i \text{ or } j > \ell_i, \forall i \in \mathcal{I} \tag{2}$$

$$y_{ij} = 0, \qquad j < e_i \text{ or } j > \ell_i + d_i - 1, \forall i \in \mathcal{I} \tag{3}$$

$$\sum_{j=e_i}^{\ell_i+d_i-1} y_{ij} = d_i, \qquad \forall i \in \mathcal{I} \tag{4}$$

$$y_{ij} - y_{ij-1} \leq x_{ij}, \qquad i \in \mathcal{I}, \forall j \in \mathcal{J} \backslash \{1\} \tag{5}$$

$$y_{i,1} \leq x_{i,1}, \qquad \forall i \in \mathcal{I} \tag{6}$$

$$\sum_{i=1}^{n} g_{ij} \left(1 - y_{ij}\right) = D_j(1+S) + r_j, \qquad \forall j \in \mathcal{J} \tag{7}$$

$$\sum_{i=1}^{n} \sum_{p=1}^{j} m'_{p,ij} x_{i,p} \leq M_j, \qquad \forall j \in \mathcal{J} \tag{8}$$

$$\sum_{i \in \mathcal{I}_k} y_{ij} \leq K_k, \qquad \forall j \in \mathcal{J}, k \in \mathcal{K} \tag{9}$$

$$x_{ij}, y_{ij} \in \{0,1\}, \qquad \forall i \in \mathcal{I}, j \in \mathcal{J} \tag{10}$$

$$r_j \geq 0, \qquad \forall j \in \mathcal{J} \tag{11}$$

Constraint (1) ensures that that each unit i has 1 starting maintenance time. Constraints (2) and (3) ensure that maintenance does not occur outside the allowable maintenance window for each unit i. Constraint (4) ensures that each unit i is under maintenance for its prescribed maintenance duration, d_i. Constraints (5) and (6) ensure that maintenance occurs over consecutive periods. Constraints (7) and (11) ensures that the load demand for each period j is met with a sufficient safety margin. Constraint (8) ensures that the maximum allowed manpower is not exceeded by the maintenance schedule. Constraint (9) ensures that the maximum number of units in each exclusion set is not exceeded.

Although this programme is not explored in this report, it provides insight into the feasible solution space of our problem.

3 Implementing Simulated Annealing

3.1 Form of Candiddate Solutions

To employ simulated annealing to the GMS problem, we denote the solution vector $\mathbf{x} = (x_1, .., x_n)$, were $x_i = j$ if maintenance on unit $i \in \mathcal{I}$ commences in period $j \in \mathcal{J}$, and 0 otherwise (as defined above).

3.2 Evaluation Function

The author quotes a number of optimality criteria for the GMS problem, namely economic, convenience, and reliability criteria. Given the success of some predecessors, the author chose to use reliability criteria, which aims to level the reserve load over the planning period. The evaluation function for minimisation was thus defined as the sum of squares of the reserve load for each week.

$$Z = \sum_{j=1}^{52} r_j$$

A penalty term was introduced in the simulated annealing algorithm to penalise solutions which did not adhere to the constraints. The penalty term is added to the evaluation of the candidate solution, thereby worsening it (since this is a minimisation problem). This approach was used instead of simply searching the feasible solution space, since it was considered more efficient. Additionally, the addition of the penalty term allows the algorithm to explore regions of the solution space which are infeasible, which may lead to better solutions down the line. The explorative nature of this approach prevents preemature convergence to local minima.

This penalty term, P was calculated as a weighted sum of the deviations of the candidate solution from the constraints on allowed maintenance window period, load, manpower, and unit exclusion (P_w, P_ℓ, P_c, P_e). That is,

$$P = w_w P_w + w_\ell P_\ell + w_c P_c + w_e P_e.$$

Hence, the final evaluation function is given by

$$Z = \sum_{j=1}^{52} r_j + P$$

3.3 Initialisation

We generate a random initial solution $\mathbf{x}_0$ by randomly sampling from the interval of earliest and latest allowed commencement period for each unit $i \in \mathcal{I}$ (e_i and ℓ_i).

Its evaluation function value is then calculated. This is encoded in the function `generateRandomSolution`.

The author proposed an improvement to this initialisation by applying a local search heuristic to the solution generated using `generateRandomSolution`. First, we generate all the possible neighbours of $\mathbf{x}_0$ using the function `createClassicalNeighbourhoodList`. Then, we find the best neighbour of $\mathbf{x}_0$, that is the neighbour with the best (lowest) evaluation function value. This neighbour is then set at the initial solution.

The initial temperature, T_0 is generated using the method proposed in [3].

$$T_0 = \frac{-\overline{\Delta E^{(+)}}}{\ln(\chi_0)}$$

Here, χ_0 is the initial acceptance ratio, which is typically set to 0.5; and $\overline{\Delta E^{(+)}}$ is the average increase in energy. This is estimated using a random walk across the solution space starting from the initial solution. The initialising of temperature is encoded in the function `initialTemperature`.

3.4 Perturbation Function

The author proposes a new perturbation function in the GMS context, namely the ejection chain move operator. The ejection move operator randomly selects a unit, and samples from its allowable starting times. Then, it finds the unit with the same starting time as the previous unit, and adjusts it by sampling from its allowable starting times. This process is repeated until the new units random starting time is the same as the first units initial starting time, or until there are no units having the starting time of the old unit.

In addition, the author has introduced hybridisation. This entails applying a local search heuristic to the incumbent solution (the best solution at a given run). The local search heuristic searches the entire neighbourhood of a solution, and finds the best neighbour. This is repeated until no further improvements are made. The search heuristic is encoded in the function `runSearchHeur`.

3.5 Cooling Schedule

The author compared a number of cooling schedules. The one that emerged as best was proposed by [4] and is defined by

$$T_{s+1} = T_s \frac{1}{1 + \frac{\ln(1+\delta)}{3\sigma_s} T_s},$$

where σ_s denotes the standard deviation of objective values up to run s, and $\delta = 0.35$ [2]. This was encoded in a function called `VLupdateT`.

However, a single run of the hybridised approach with this cooling schedule incurred hours of runtime in our interpretation of the algorithm in `R`. It was hence decided to employ a simpler cooling schedule. The geometric variant was chosen, which significantly reduced the runtime. This cooling schedule is defined by

$$T_{s+1} = \alpha T_s,$$

where $\alpha \in (0, 1)$. The convention is to use a value of α between 0.8 and 0.99 [1]. For our replication, a value of 0.9 was used.

3.6 Termination Criteria

The inner loop terminates when the number of solutions attempted exceeds $100N$, or the number of solutions accepted exceeds $12N$, where N = degrees of freedom, which in this case is n. The outer loop terminates when the temperature reaches T_{min}, or if Ω_{frozen} successive loops occur without acceptance of a solution. Both the parameters T_{min} and Ω_{frozen} are user-defined. In other words, the algorithm terminates when a sufficiently low temperature is reached, or if the algorithm has made no improvement on the solution for "enough" runs.

4 Data and Parameters

The data was provided in [2] in tables which can be found in Appendix 8.1. This had to be inputted manually into `R`. The columns containing capacity, earliest starting time, latest starting time, and duration could be easily stored in a dataframe. However, since the manpower requirement vectors are of different lengths for each unit, these were stored in a list for each system. The exclusion constraints for the 32-unit system were also stored in a list, with I_k defined as the vector of units for each exclusion set k in Figure 5. K_k is defined as the maximum units allowed in operation for $i \in I_k$, and is given by the last column of the table. In addition to the exclusion constraints, further constraints on the maximum number of units allowed in operation for a period j, M_j. In the 32-unit system, this was 25 for all $j \in \mathcal{J}$. For the 21-unit system this was 20 for all $j \in \mathcal{J}$. The demand for the 21-unit system was consistent across the planning period at 4738 MW, and that of the 32-unit system is varying and shown in Figure 6.

The author provided additional information that the values of S were 0.15 and 0.2 for the 32- and 21-unit test system respectively. Additionally, the author used a value of $\chi_0 = 0.5$ for the generation of initial temperature, and a value of $\delta = 0.35$ for the cooling schedule presented by Van Laarhoven [4].

The author derived the weights assigned to constraint set penalties using test runs and determining which weights produced a good ratio of feasible to infeasible solutions. The aim of the algorithm is not to avoid infeasible solutions entirely, but rather to allow them to be accepted in some cases to avoid the algorithm settling into a local minima. The weights used for the 32-unit test system were $P_w = 40000, P_\ell = 1, P_c = 20000, P_e = 20000$. The weights used for the 21-unit test system were $P_w = 500000, P_\ell = 1, P_c = 20000, P_e = 0$, since there were no exclusion constraints outlined for the 21-unit system.

Since no information was found regarding the maintenance window extension parameter, W_{ext}, a value of 2 was used. It was decided to use this value instead of 0 so that the algorithm could explore infeasible regions of the solution space. Additionally, the values of $T_{min} = 1$ and $\Omega_{frozen} = 100$ were assumed for the termination criteria. The author ran the hybridised algorithm on each test system 50 times. Despite all efforts to reduce runtime (including parallelisation, vectorisation where possible), this was not feasible for our rendition. Instead, three runs were performed on each test system.

5 Coding problem in `R`

5.1 Overview

The experiments outlined in the paper were run in MatLab, however this interpretation will attempt to replicate it in `R`. The functions were defined according to the pseudo-code presented in Chapter 4 [2], and replicated for this report using the original names. The hybrised approach was used, since this produced the best solutions. This approach is briefly summarised in Figure 1.

A function called `createEjectionChainList` was outlined by the author which takes a candidate solution and produces an ejection chain matrix, with the first column containing the units for replacement, and the second containing the replacement value. Another function called `checkFeasibilityAndCalculatePenalty` was outlined. This function calculates P. All the possible moves in the classical neighbourhood move operator are generated in the function `createClassicalNeighbourhoodList`. A function to generate a random solution and calculate its objective value was encoded, `generateRandomSolution`. This was then called in `generateGoodRandomSolution` - another function that generates a random solution and its objective value, however with some improvements. The improvements arise from the application of the local search heuristic. This is en-

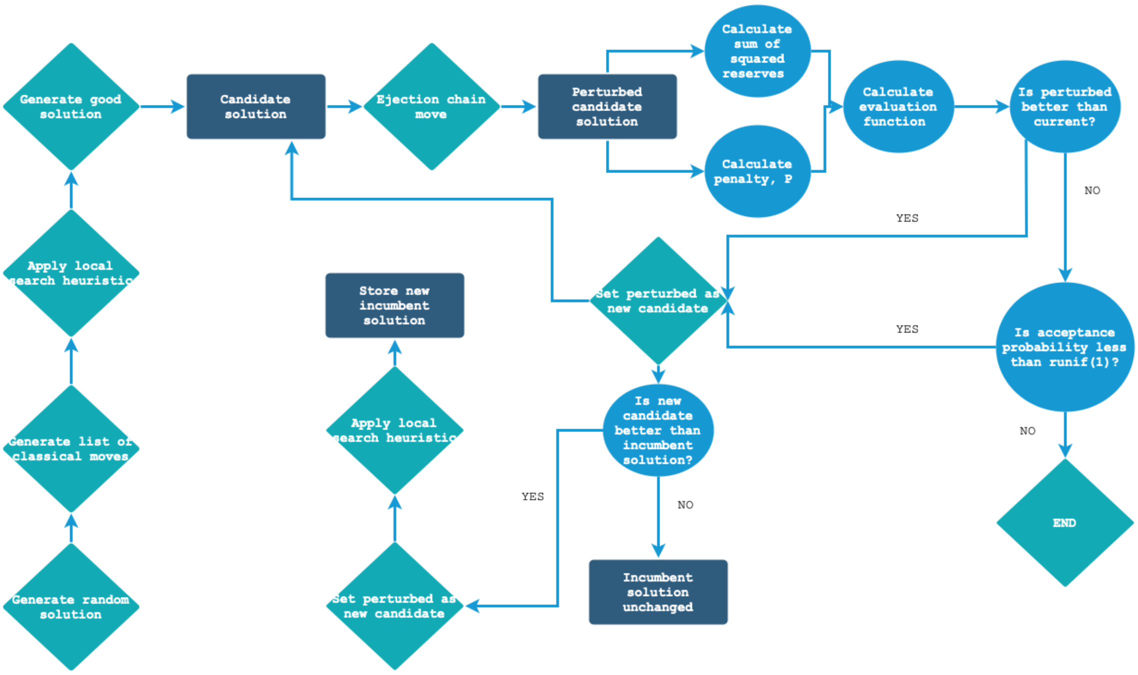

Figure 1: Scheme of hybridised approach

coded in the function `runSearchHeur`, which finds the best neighbour from the `createClassicalNighbourhoodList` output and replaces it as the incumbent solution if it is better than the current incumbent solution.

Some additional functions were created for the implementation of the methodology. Namely, a function called `applyChain` was created to apply an ejection chain outputted by `createEjectionChainList` [2] to a solution $\mathbf{x}$, which outputs a new solution. Lastly, a function was created to excecute the evaluation function, namely `calcObjVal`.

5.1.1 Sampling from allowable starting times for unit i

The author refers to sampling from the allowable maintenance starting times $[e_i - W_{ext}, \ell_i + W_{ext}]$ in some of their algorithms. However this does not account for the duration of maintenance d_i. If $\ell_i + W_{ext}$ is too large, the entire duration may not fit in the 52 week planning period. Additionally, we may encounter $e_i - W_{ext} < 1$, which is of course outside the maintenance planning period. To avoid this issue, at each step where we are required to sample from $[e_i - W_{ext}, \ell_i + W_{ext}]$, we instead sample from $[\max(1, e_i - W_{ext}), \min(\ell_i + W_{ext}, 52 - d_i + 1]$.

5.2 Definitions of Functions

In this section the functions which are not inherited from Chapter 4 [2] will be defined. The code corresponding to these definitions can be found in Appendix 8.2.

5.2.1 `trdata(x, data, M)`

Input: Candidate solution, all data in dataframe, list of vectors of manpower upperbound within duration of maintenance for each unit.

Output: A list of matrices $X = \{x_{ij}\}, Y = \{y_{ij}\}, M_m = \{m_{ij}\}, G_y = \{y_{ij} \times g_{ij}\}, \forall i \in \mathcal{I}, j \in \mathcal{J}$

1. Initialise X and Y as matrices of dimension $n \times m$ containing zeroes.
2. for each $i \in \{1, .., n\}$
   (a) Set the $\{i, x_i\}^{th}$ element of X to 1.
   (b) Set i^{th} row of Y to have 1 in each element from $j = x_i$ to $j = x_i + d_i - 1$.
3. Initialise M_m as matrix of dimension $n \times m$ containing zeroes.
4. Initialise G as matrix of dimension $n \times m$ with each column as the generator capacity for unit i as presented in Figure 4 or 7.
5. for each $i \in \{1, ..., n\}$
   (a) for each j where $y_{ij} = 1$ in Y
      i. Set m_{ij} in M_m to the j^{th} element of the i^{th} item in the list M.
   (b) for each j where $y_{ij} = 0$ in Y
      i. Set g_{ij} in G to 0. That is, set generating capacity to 0 during maintenance.
   (c) Return a list of X, Y, M_m, G_y.

5.2.2 `checkFeasibilityAndCalculatePenalty(x, data, W, D, Mjs, KK, M)`

Although this function is described in [2], some adjustments were made for partial vectorisation of the calculation using the matrices obtained using `trdata`. This was done in the interest of reducing runtime vs. using equivalent if and/or for loops in `R`.

Input: Candidate solution, all data in dataframe, vector of weights for each constraint set, load demand for each time period, vector of overall manpower upperbound, list of of exclusion subsets I_k and corresponding $K_k \forall k \in \mathcal{K}$, list of vectors of manpower upperbound within duration of maintenance for each unit.

Output: Total penalty of solution $\mathbf{x}$, P.

1. Initialise $P_w, P_\ell, P_c, P_e = 0$.
2. Generate X, Y, M_m, G_y
   (a) Apply `trdata` to `(x, data, M)`

   Maintenance window penalty

3. for $i \in \{1, ..., n\}$, calculate maintenance window penalty, P_w^i for each x_i as follows:
   (a) if $x_i \notin [e_i, \ell_i]$
      i. if $x_i < e_i$
         A. $P_w^i = e_i - x_i$
      ii. else
         A. $P_w^i = x_i - \ell_i$
   (b) Add to P_w by setting $P_w = P_w + w_w P_w^i$

   Load demand penalty

4. Take `colSums` of G_y to get the total load for each time period $j \in \mathcal{J}$. From this vector subtract the vector $\mathbf{D} = \{Dj\}$ element-wise to obtain $r_j \forall j \in \mathcal{J}$ in a vector called `rj`.
5. for $j \in \{1, .., m\}$ calculate the load demand penalty P_ℓ^j.
   (a) Calculate $P_\ell^j = \max\{-r_j, 0\}$
   (b) Add to P_ℓ by setting $P_\ell = P_\ell + P_\ell^j$

   Crew constraint penalty

6. Calculate $\sum_{i=1}^{n} m_{i,j} y_{i,j} - M_j \forall j \in \mathcal{J}$ by taking `colSums` of M_m, and subtracting $\mathbf{M_j}$ element-wise. Store in a vector $\mathbf{v}$.
7. for $j \in \{1, .., m\}$
   (a) Calculate $P_c^j = \max\{v_j, 0\}$
   (b) Add to P_c by setting $P_c = P_c + P_c^j$

Exclusion penalty

8. for $k \in \{1, ..., K\}$
   (a) Extract I_k and K_k from the list `KK`
   (b) Calculate $\sum_{i \in \mathcal{I}_k} y_{ij}$ (the number of units in exclusion subset under operation) by taking the colSums of the rows in Y corresponding to units in I_k.
   (a) for $j \in \{1, .., m\}$
      i. Calculate $P_e^{k,j} = \max\left\{\sum_{i \in \mathcal{I}_k} y_{i,j} - K_k, 0\right\}$
      ii. Add to P_e by setting $P_e = P_e + P_e^{k,j}$
9. Return $P = P_w + w_\ell P_\ell + w_c P_c + w_e P_e$, where the weights are obtained from `W`.

Note that although the 21-unit test system had no exclusion constraints, this function could be used by setting $\mathcal{K} = \{1\}$, $I_k = \mathcal{I}$ with $K_k = 21$. This allows all units to be under maintenance at any time.

5.2.3 `calcObjVal(x, data, D, M)`

Input: Candidate solution, all data in data frame, vector of load demand for each time period, list of vectors of manpower upperbound within duration of maintenance for each unit.

Output: $\sum_{j=1}^{m} r_j$ for a candidate solution $\mathbf{x}$

1. Extract vector of generating capacity $\mathbf{g}$ from `data`.
2. Obtain Y
   (a) Pass `(x, data, M)` to `trdata`
3. Initialise sum of all reserves (`totsum`) to 0.
4. for $j \in \{1, .., m\}$ find r_j^2
   (a) Calculate $D_j(1 + s)$
   (b) Initialise $\sum_{i=1}^{n} g_{ij}(1 - y_{ij})$ as `t1sum= 0`
   (c) for $i \in \{1, .., n\}$
      i. Extract g_{ij}, y_{ij} from G_y, Y respectively.
      ii. Add $g_{ij}(1 - y_{ij})$ to `t1sum`

   (d) Calculate r_j by subtracting demand with safety window from the total generating capacity `t1sum`, and square to get r_j^2.

   (e) Add r_j^2 to `totsum`.

5. Return `totsum`

5.2.4 `applyChain(x, Chain)`

Input: Candidate solution, $c \times 2$ matrix obtained from `createEjectionChainList` (c random and dependent on chain).

Output: Transformed solution `x'` with `Chain` applied

1. Set `newx` as original `x`
2. for $i \in \{1, .., c\}$
   (a) Define `unit` as i^{th} entry from first column of `Chain`. This is the unit to be changed.
   (b) Define `time` as i^{th} entry from second column of `Chain`. This is the time to assign to `unit`.
   (c) Set the i^{th} entry of `newx` to `time`.
3. Return `newx`

5.2.5 `is.feasible(x, data, W, D, Mjs, KK, M)`

Input: Candidate solution, dataframe, vector of weights, vector of upperbounds on manpower, list of of exclusion subsets I_k and corresponding $K_k \forall k \in \mathcal{K}$, list of vectors of manpower upperbound within duration of maintenance for each unit

Output: `TRUE` or `FALSE` indicating whether candidate solution is feasible

1. Calculate P by passing `(x, data, W, D, Mjs, KK, M)` to `checkFeasibilityAndCalculatePenalty`.
2. Define `feas` as the outcome of a logical test of if $P = 0$
3. Return `feas`

6 Results

The code which used to obtaine the results can be found in Appendix 8.2. The progression of the algorithm for the 21- and 32-unit test systems are presented in

Figures 7 and 3 respectively. The algorithm converges in all cases. The search of the solution space appears convincing in both cases. However, this does not necessarily imply global minima were reached. The solution vectors and evaluation function values for the 21- and 32-unit test system runs are presented in Tables 1 and 2. The solutions obtained were all feasible, however, the hybridised algorithm was run only three times. Furthermore, the values are not consistent with the results obtained in the paper. This may be due to the assumption of certain settings (such as W_{ext}).

Table 1: Results from three runs of 21-unit test system

	Cand.	Inc.	Cand.	Inc.	Cand.	Inc.
x_1	16	19	18	19	5	9
x_2	48	45	27	31	48	33
x_3	23	21	5	13	8	1
x_4	2	6	26	4	25	26
x_5	27	32	45	48	31	45
x_6	8	14	7	15	12	23
x_7	5	7	10	10	17	16
x_8	32	39	34	41	37	27
x_9	3	9	9	6	14	15
x_{10}	15	1	2	22	10	4
x_{11}	1	26	25	1	24	3
x_{12}	44	50	30	36	40	49
x_{13}	19	10	22	8	22	19
x_{14}	11	2	1	18	4	6
x_{15}	21	12	15	5	15	21
x_{16}	25	24	13	2	20	11
x_{17}	39	31	44	47	27	40
x_{18}	37	37	50	39	29	38
x_{19}	47	49	52	44	28	52
x_{20}	41	27	40	27	43	41
x_{21}	13	17	19	24	2	13
Eval.	244379	239440	247550	309098	444986	239758

Table 2: Results from three runs of hybridised approach for 32-unit test system

	Cand.	Inc.	Cand.	Inc.	Cand.	Inc.
x_1	21	5	3	3	14	14
x_2	23	1	1	1	7	1
x_3	4	3	20	20	3	3
x_4	40	44	42	43	30	27
x_5	22	13	3	3	1	7
x_6	32	45	33	33	45	45
x_7	1	22	6	6	4	4
x_8	44	30	43	42	27	30
x_9	29	27	27	27	41	41
x_{10}	27	40	17	17	42	42
x_{11}	16	20	30	30	20	20
x_{12}	9	9	8	15	9	6
x_{13}	15	6	15	8	6	9
x_{14}	36	34	37	37	37	37
x_{15}	14	23	16	16	1	23
x_{16}	30	2	1	1	1	1
x_{17}	17	18	36	36	24	24
x_{18}	35	19	24	7	21	21
x_{19}	41	11	1	1	22	22
x_{20}	6	16	4	4	16	16
x_{21}	34	36	35	35	34	34
x_{22}	10	10	9	9	10	10
x_{23}	38	38	31	31	31	31
x_{24}	8	7	21	21	28	28
x_{25}	3	35	22	21	8	21
x_{26}	7	4	34	34	21	37
x_{27}	12	25	14	14	13	13
x_{28}	13	8	11	11	12	12
x_{29}	26	13	41	41	36	8
x_{30}	19	15	13	13	15	15
x_{31}	25	26	26	26	26	26
x_{32}	31	31	38	38	38	38
Eval.	11149975	11142769	11114188	11113055	11060450	11057430

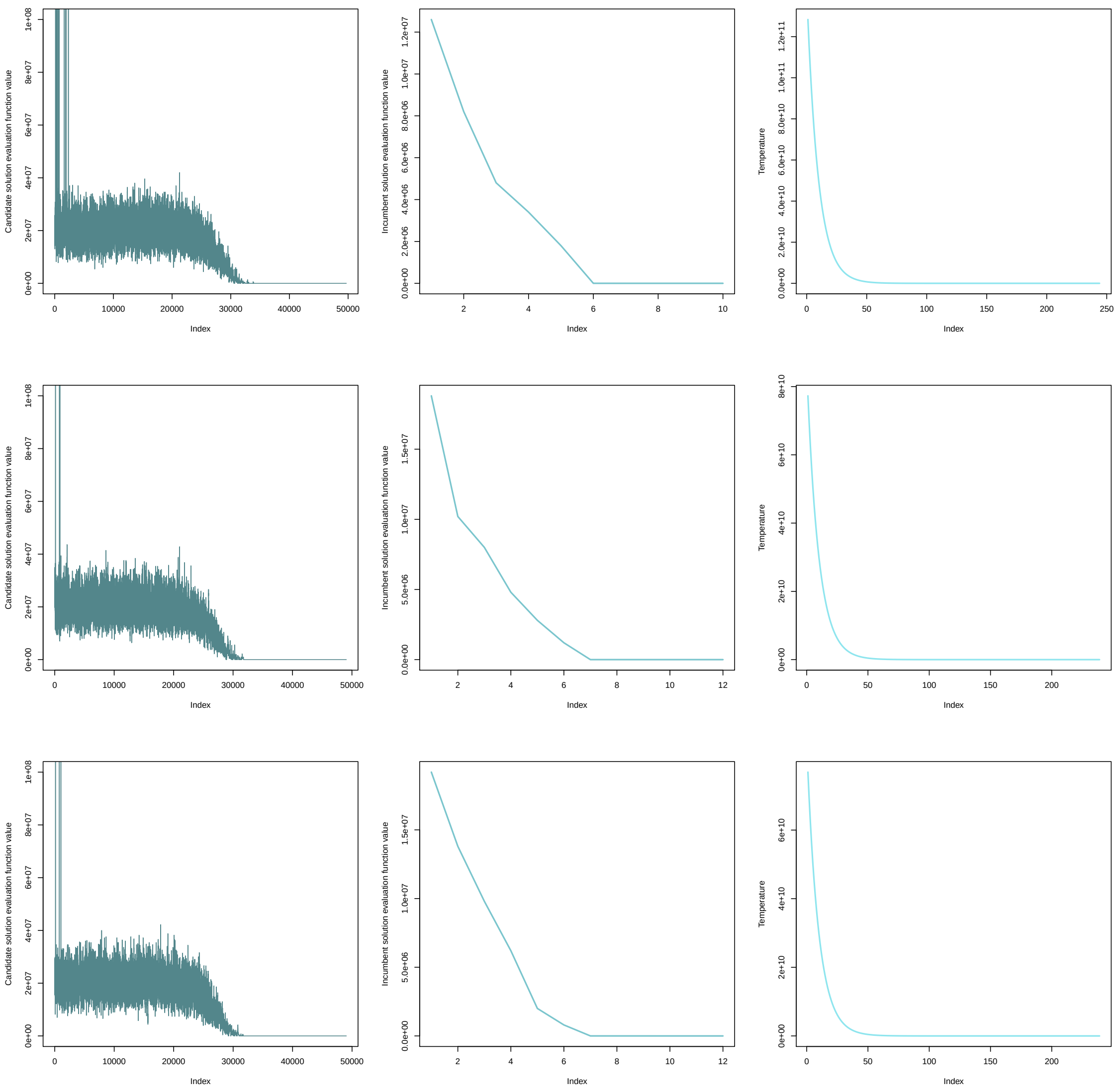

Figure 2: Plots of accepted evaluation function value (including inner loop), incumbent evaluation value, and temperature for three runs of hybridised approach on 21-unit test system

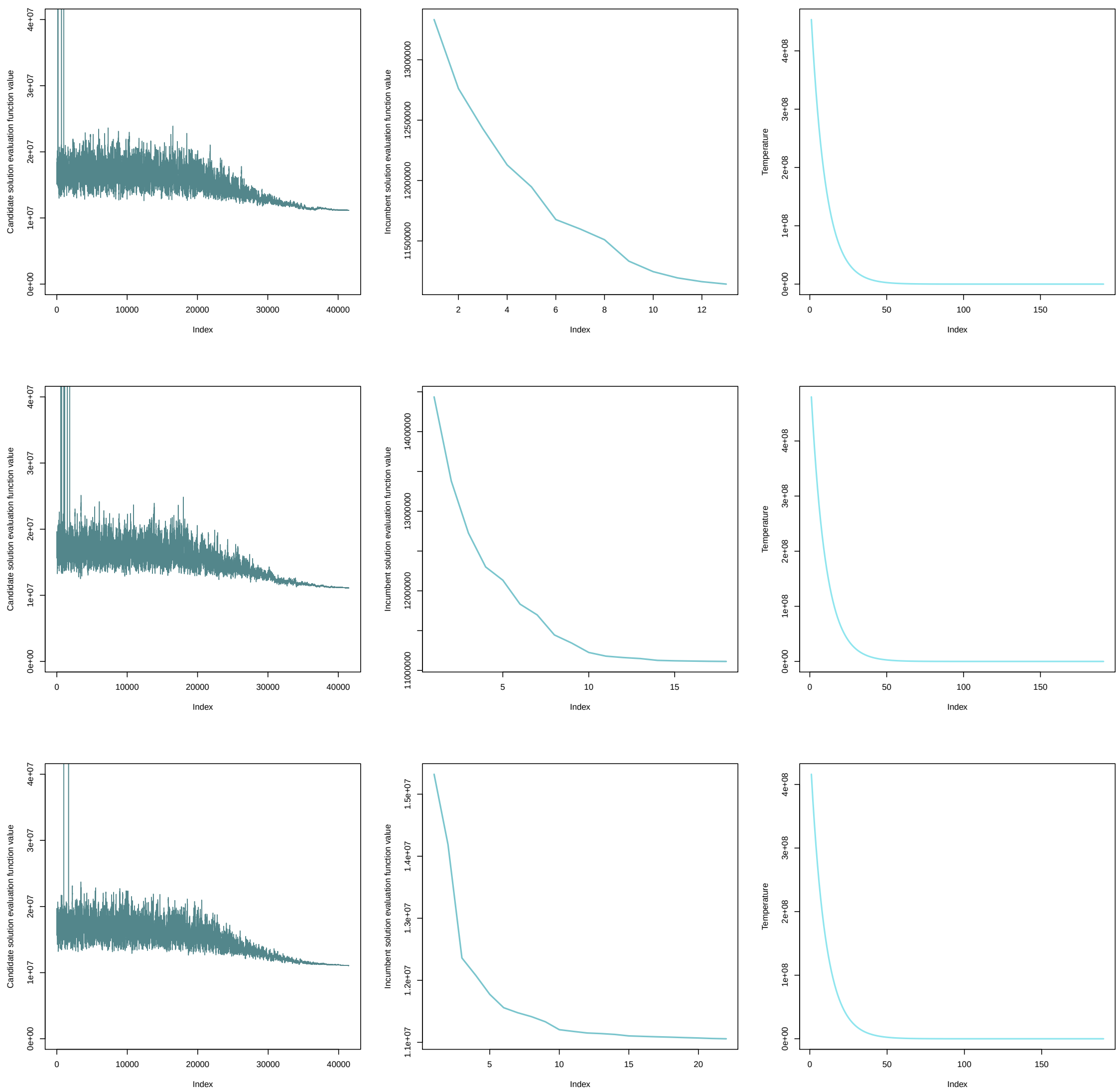

Figure 3: Plots of accepted evaluation function value (including inner loop), incumbent evaluation value, and temperature for three runs of hybridised approach on 32-unit test system

7 Discussion and Conclusion

We could not reproduce the exact results obtained by the author, due to a number of factors. First, it should be noted that there is a random component to the algorithms, which will cause variation. However, under circumstances such as this where the algorithm is run a few times, the results should be similar. Another factor contributing to this is the extensive run times of our algorithm, even when employing multiple cores and a simpler cooling schedule. Perhaps this could be attributed to the use of `R` instead of MatLab, or inefficiencies in the coding of the problem in `R`.

The advantage of the hybridised approach proposed by the author is offset by these extensive run times in our replication. Although better solutions are obtained compared to the approach that is status quo, it may be the case that simpler approaches are more efficient. The author quotes simplicity and low computation time as advantages of using heuristic techniques to solve the GMS problem, however these are not true for the hybridised approach, at least in our replication.

References

[1] Richard W Eglese. Simulated annealing: a tool for operational research. *European journal of operational research*, 46(3):271–281, 1990.

[2] EB Schlünz and JH Van Vuuren. An investigation into the effectiveness of simulated annealing as a solution approach for the generator maintenance scheduling problem. *International Journal of Electrical Power & Energy Systems*, 53:166–174, 2013.

[3] Eric Triki, Yann Collette, and Patrick Siarry. A theoretical study on the behavior of simulated annealing leading to a new cooling schedule. *European Journal of Operational Research*, 166(1):77–92, 2005.

[4] PJM Van Laarhoven and EHL Aarts. Simulated annealing: theory and applications. dordrecht: D. *Reidel Pub. Comp., Netherlands*, 1987.

8 Appendix

8.1 Data

Unit	Capacity (MW)	Earliest starting time (week)	Latest starting time (week)	Duration (weeks)	Manpower required during each week of maintenance
1	20	1	25	2	7, 7
2	20	1	25	2	7, 7
3	76	1	24	3	12, 10, 10
4	76	27	50	3	12, 10, 10
5	20	1	25	2	7, 7
6	20	27	51	2	7, 7
7	76	1	24	3	12, 10, 10
8	76	27	50	3	12, 10, 10
9	100	1	50	3	10, 10, 15
10	100	1	50	3	10, 10, 15
11	100	1	50	3	15, 10, 10
12	197	1	23	4	8, 10, 10, 8
13	197	1	23	4	8, 10, 10, 8
14	197	27	49	4	8, 10, 10, 8
15	12	1	51	2	4, 4
16	12	1	51	2	4, 4
17	12	1	51	2	4, 4
18	12	1	51	2	4, 4
19	12	1	51	2	4, 4
20	155	1	23	4	5, 15, 10, 10
21	155	27	49	4	5, 15, 10, 10
22	400	1	21	6	15, 10, 10, 10, 10, 5
23	400	27	47	6	15, 10, 10, 10, 10, 5
24	50	1	51	2	6, 6
25	50	1	51	2	6, 6
26	50	1	51	2	6, 6
27	50	1	51	2	6, 6
28	50	1	51	2	6, 6
29	50	1	51	2	6, 6
30	155	1	23	4	12, 12, 8, 8
31	155	1	49	4	12, 12, 8, 8
32	350	1	48	5	5, 10, 15, 15, 5

Figure 4: Data for 32-unit test system [2]

Exclusion set	Units	Maximum
1	1, 2, 3, 4	2
2	5, 6, 7, 8	2
3	9, 10, 11	1
4	12, 13, 14	1
5	15, 16, 17, 18, 19, 20	3
6	24, 25, 26, 27, 28, 29	3
7	30, 31, 32	1

Figure 5: Exclusion subsets for 32-unit test system [2] (K, I_k, K_k)

Week	Demand (MW)	Week	Demand (MW)	Week	Demand (MW)	Week	Demand (MW)
1	2 457	14	2 138	27	2 152	40	2 063
2	2 565	15	2 055	28	2 326	41	2 118
3	2 502	16	2 280	29	2 283	42	2 120
4	2 377	17	2 149	30	2 508	43	2 280
5	2 508	18	2 385	31	2 058	44	2 511
6	2 397	19	2 480	32	2 212	45	2 522
7	2 371	20	2 508	33	2 280	46	2 591
8	2 297	21	2 440	34	2 078	47	2 679
9	2 109	22	2 311	35	2 069	48	2 537
10	2 100	23	2 565	36	2 009	49	2 685
11	2 038	24	2 528	37	2 223	50	2 765
12	2 072	25	2 554	38	1 981	51	2 850
13	2 006	26	2 454	39	2 063	52	2 713

Figure 6: Weekly demand load for the 32-unit system

Unit	Capacity (MW)	Earliest starting time (week)	Latest starting time (week)	Duration (weeks)	Manpower required during each week of maintenance
1	555	1	20	7	10, 10, 5, 5, 5, 5, 3
2	555	27	48	5	10, 10, 10, 5, 5
3	180	1	25	2	15, 15
4	180	1	26	1	20
5	640	27	48	5	10, 10, 10, 10, 10
6	640	1	24	3	15, 15, 15
7	640	1	24	3	15, 15, 15
8	555	27	47	6	10, 10, 10, 5, 5, 5
9	276	1	17	10	3, 2, 2, 2, 2, 2, 2, 2, 2, 3
10	140	1	23	4	10, 10, 5, 5
11	90	1	26	1	20
12	76	27	50	3	10, 15, 15
13	76	1	25	2	15, 15
14	94	1	23	4	10, 10, 10, 10
15	39	1	25	2	15, 15
16	188	1	25	2	15, 15
17	58	27	52	1	20
18	48	27	51	2	15, 15
19	137	27	52	1	15
20	469	27	49	4	10, 10, 10, 10
21	52	1	24	3	10, 10, 10

Figure 7: Data for 21-unit test system [2]

8.2 R Code

```
set.seed(2020)
################# DATA ###################
##### 21-unit test system data
data_21_unit <- c(1, 555, 1, 20,
                  2, 555, 27, 48,
                  3, 180, 1, 25,
                  4, 180, 1, 26,
                  5, 640, 27, 48,
                  6, 640, 1, 24,
                  7, 640, 1, 24,
                  8, 555, 27, 47,
                  9, 276, 1, 17,
                  10, 140, 1, 23,
                  11, 90, 1, 26,
                  12, 76, 27, 50,
                  13, 76, 1, 25,
                  14, 94, 1, 23,
                  15, 39, 1, 25,
                  16, 188, 1, 25,
                  17, 58, 27, 52,
                  18, 48, 27, 51,
                  19, 137, 27, 52,
                  20, 469, 27, 49,
                  21, 52, 1, 24
)

data21 <- matrix(data_21_unit, nrow = 21,
                 ncol = 4, byrow = T)
colnames(data21) <- c("unit", "G", "E", "L")
duration <- c(7,5,2,1,5,3,3,6,10,4,1,3,2,4,
              2,2,1,2,1,4,3)
data21 <- cbind (data21, duration)
# exclusion - max 20 from among all units
KK21 <- list(list(c(1:21), 20))
# manpower
M21 <- list(c(10,10,5,5,5,5,3),
            c(10,10,10,5,5),
            c(15,15),
            c(20),
```

```
            c(10,10,10,10,10),
            c(15,15,15),
            c(15,15,15),
            c(10,10,10,5,5,5),
            c(3,2,2,2,2,2,2,2,2,3),
            c(10,10,5,5),
            c(20),
            c(10,15,15),
            c(15,15),
            c(10,10,10,10),
            c(15,15),
            c(15,15),
            c(20),
            c(15,15),
            c(15),
            c(10,10,10,10),
            c(10,10,10))
# make a data frame
data21 <- as.data.frame(data21)
# load demand - constant throughout 52 week period
D21 <- rep(4739, 52)
# safety margin
S21 <- 0.2
# penalty weights
# window, load, crew
W21 <- c(500000, 1, 200000, 0)
# 20 crew members per week
Mj21 <- rep(20,52)
# exclusion - no exclustions
# can choose up to 21 of entire set
# (created for generalisation)
KK21 <- list(list(c(1:21), 21))

####################################
### 32-unit test system data
# manpower
M32 <- list(c(7,7),
            c(7,7),
            c(12,10,10),
            c(12,10,10),
```

```
            c(7,7),
            c(7,7),
            c(12,10,10),
            c(12,10,10),
            c(10,10,15),
            c(10,10,15),
            c(15,10,10),
            c(8,10,10,8),
            c(8,10,10,8),
            c(8,10,10,8),
            c(4,4), c(4,4), c(4,4), c(4,4), c(4,4),
            c(5,15,10,10), c(5,15,10,10),
            c(15,10,10,10,10,5), c(15,10,10,10,10,5),
            c(6,6), c(6,6), c(6,6), c(6,6), c(6,6), c(6,6),
            c(12,12,8,8), c(12,12,8,8),
            c(5,10,15,15,5))

# 25 crew members per week
Mj32 <- rep(25, 52)

# 32-unit test system data
data_32_unit <- matrix(c(1, 20, 1, 25, 2,
                         2, 20, 1, 25, 2,
                         3, 76, 1, 24, 3,
                         4, 76, 27, 50, 3,
                         5, 20, 1, 25, 2,
                         6, 20, 27, 51, 2,
                         7, 76, 1, 24, 3,
                         8, 76, 27, 50, 3,
                         9, 100, 1, 50, 3,
                         10, 100, 1, 50, 3,
                         11, 100, 1, 50, 3,
                         12, 197, 1, 23, 4,
                         13, 197, 1, 23, 4,
                         14, 197, 27, 49, 4,
                         15, 12, 1, 51, 2,
                         16, 12, 1, 51, 2,
                         17, 12, 1, 51, 2,
                         18, 12, 1, 51, 2,
                         19, 12, 1, 51, 2,
```

```
                         20, 155, 1, 23, 4,
                         21, 155, 27, 49, 4,
                         22, 400, 1, 21, 6,
                         23, 400, 27, 47, 6,
                         24, 50, 1, 51, 2,
                         25, 50, 1, 51, 2,
                         26, 50, 1, 51, 2,
                         27, 50, 1, 51, 2,
                         28, 50, 1, 51, 2,
                         29, 50, 1, 51, 2,
                         30, 155, 1, 23, 4,
                         31, 155, 1, 49, 4,
                         32, 350, 1, 48, 5), nrow=32, ncol=5, byrow=T)
colnames(data_32_unit) <- colnames(data21)
data32 <- as.data.frame(data_32_unit)

# exclusion subsets
KK32 <- list(list(c(1,2,3,4),2),
             list(c(5,6,7,8), 2),
             list(c(9, 10, 11), 1),
             list(c(12, 13, 14), 1),
             list(c(15, 16, 17, 18, 19, 20), 3),
             list(c(24, 25, 26, 27, 28, 29), 3),
             list(c(30, 31, 32), 1))
# demand
D32 <- matrix(c(1, 2457, 14, 2138, 27, 2152, 40, 2063,
                2, 2565, 15, 2055, 28, 2326, 41, 2118,
                3, 2502, 16, 2280, 29, 2283, 42, 2120,
                4, 2377, 17, 2149, 30, 2508, 43, 2280,
                5, 2508, 18, 2385, 31, 2058, 44, 2511,
                6, 2396, 19, 2480, 32, 2212, 45, 2522,
                7, 2371, 20, 2508, 33, 2280, 46, 2591,
                8, 2296, 21, 2440, 34, 2078, 47, 2679,
                9, 2109, 22, 2311, 35, 2069, 48, 2537,
                10, 2100, 23, 2565, 36, 2009, 49, 2685,
                11, 2038, 24, 2528, 37, 2223, 50, 2765,
                12, 2072, 25, 2554, 38, 1981, 51, 2850,
                13, 2006, 26, 2454, 39, 2063, 52, 2713), ncol=8, byrow = T)
D32 <- c(D32[,-c(1,3,5,7)])
# Safety margin
```

```
S32 <- 0.15
# penalty weights
W32 <- c(40000, 1, 20000, 20000)

#### FUNCTIONS ##########
### function that updates T
### Van Laarhoven
VLupdateT <- function(Temp, Z, g){
  # input: Ts
  # Z = obj value functions so far
  # g = small number
  # Output: Ts+1

  # sd of obj values
  sigs <- sd(Z)

  frac <- (log(1+g)/(3*sigs))*Temp

  # Ts+1
  val <- Temp*(1/(1 + frac))

  return(val)
}

### geometric
GupdateT <- function(Temp, alpha=0.9){
  newT <- alpha*Temp
  return(newT)
}

### function to transform data to matrices
trdata <- function(x, data, M){
  # Input:
  # x = possible solution in xi form,
  # data = data pf test set
  # m = total number of time periods
  # Ouput:
  # X = xij maintenance schedule
  # Y = yij schedule
  # M = matrix of mij's
```

```
  # G = matrix of gij's

  # total units
  n <- nrow(data)

  # set up desired matrices
  X <- matrix(0, ncol=52, nrow=n)
  Y <- X

  # transfrom xi's to X
  for(i in 1:length(x)){
    X[i,x[i]] <- 1
    # use duration to get Y from xi = start
    Y[i, (x[i]:min(52,x[i]+data$duration[i]-1))] <- 1
  }

  # init matrix for output
  # Manpower matrix of mij's
  MM <- matrix(0, ncol = 52, nrow = nrow(data))
  # Generating capacity matrix of gij's
  GG <-matrix(data$G, ncol=52, nrow = length(x), byrow = F)
  # populate
  for (i in 1:length(x)){
    # ith row of Y
    yi <- Y[i,]

    # populate
    MM[i,as.logical(yi)] <- M[[i]][1:sum(yi)]
    GG[i,as.logical(yi)] <- 0
  }

  # return both X and Y
  return(list(X=X, Y=Y, M=MM, G=GG))
}

## function to generate ejection chain - FINAL
createEjectionChainList <- function(Unit, E, L, wext, x, duration){
  #Input:
  # i = The unit at the head of an ejection chain (randomly selected ),
  # n = the number of units,
```

```
# e,l = the vectors containing the earliest and latest maintenance starting time
# wext = the maintenance commencement extension parameter,
# x = the current solution vector
# Output: chain

# init chain
chain <- matrix(NA, ncol=2, nrow=1)

# Possible times to start unit
upper <- min(L[Unit] + wext, 52 - duration[Unit] + 1)
lower <- max(1, E[Unit] - wext)
possibleTimes <- c(lower:upper)
# remove x[Unit] if in possibleTimes
if(x[Unit] %in% possibleTimes){
  possibleTimes <- possibleTimes[-which(possibleTimes==x[Unit])]
}

# new value for x[Unit]
newTime <- sample(rep(possibleTimes,2),1)

# set counter
counter <- 1

# store
chain[counter, ] <- c(Unit, newTime)

# which units to choose from
remainingUnits <- c(1:length(x))[-Unit]

# Begin loop
notDone <- TRUE
while(notDone==T){
  # init potential units to choose from
  potentialUnits <- c()
  for(i in remainingUnits){
    # check which units share time with x[Unit]
    if(x[i]==newTime){
      potentialUnits <- c(potentialUnits, i)
    }
  }
```

```
    # if there are units that share time with x[Unit]
    if(length(potentialUnits) > 0){
      # set counter for storage
      counter <- counter + 1

      # randomly choose unit from potentials
      newUnit <- sample(rep(potentialUnits,2), 1)

      # possible starting times for newUnit
      upper <- min(L[newUnit] + wext, 52 - duration[newUnit] + 1)
      lower <- max(E[newUnit] - wext, 1)
      possibleTimes <- c(lower:upper)
      if(x[newUnit] %in% possibleTimes){
        possibleTimes <- possibleTimes[-which(possibleTimes==x[newUnit])]
      }

      # randomly select new time
      newTime <- sample(rep(possibleTimes,2), 1)

      # store newUnit, newTime
      chain <- rbind(chain, c(newUnit, newTime))

      # remove new unit from remaining units
      remainingUnits <- c(1:length(x))[-newUnit]

      # check if returned to first unit
      if(newTime==chain[1,2]){notDone <- FALSE}
    }else{
      # if there are no potential units, end
      notDone <- FALSE
    }
  }

  return(chain)
}

# function to calc P
checkFeasibilityAndCalculatePenalty <- function(x, data, W, D, Mjs, KK, M){
  # Input:
```

```
# x = The current solution vector,
# data = the problem's full dataset
# W = vector of weights
# D = demand
# Mjs = vector of overall manpower upperbound
# KK = set of exclusion subsets
# M = manpower upperbound list
# Output: The total penalty term for the current solution

# init penalties for each constraint set
Pw <- 0
Pl <- 0
Pc <- 0
Pe <- 0

# required matrices
X <- trdata(x = x, data = data, M=M)$X
Y <- trdata(x = x, data = data, M=M)$X
M <- trdata(x = x, data = data, M=M)$M
G <- trdata(x = x, data = data, M=M)$G


# Maintenance window
for(i in 1:length(x)){
  ei <- data$E[i]
  li <- data$L[i]
  wi <- W[1]

  if(!(x[i] %in% (ei:li))){
    if(x[i] < ei){pwi <- ei-x[i]}else{pwi <- x[i] - li}
    # add pwi*weight
    Pw <- Pw + wi*pwi
  }
}


# Load constraints
rjs <- colSums(G) - D
for (rj in rjs){
  # find penalty for time period j
```

```
    plj <- max(-rj, 0)
    # add to total load penalty
    Pl <- Pl + plj
  }


  # Crew constraints
  vals <- colSums(M) - Mjs
  for(v in vals){
    pcj <- max(0, v)
    # add to sum
    Pc <- Pc + pcj
  }


  # Exclusion constraints
  for(k in 1:length(KK)){
    # exclusion subset
    Ik <- unlist(KK[[k]][1])
    # max units allowed
    Kk <- unlist(KK[[k]][2])
    # total units in Ik in operation for each j
    sumyij <- colSums(Y[Ik,])

    for (j in 1:52){
      # penalise if +ve deviation
      if(sumyij[j] - Kk >0){
        pekj <- sumyij[j] - Kk
        # add to total penalisation sum
        Pe <- Pe + pekj
      }
    }
  }

  # return weighted sum
  return(sum(W*c(Pw, Pl, Pc, Pe)))
}

# function that creates list of classical moves
createClassicalNeighbourhoodList <- function(n, E, L, wext, duration){
```

```
  #Input:
  # n= The number of units,
  # E, L =  the vectors containing the earliest and latest maintenance
  # starting times for all units, the maintenance commencement extension
  # parameter
  # duration = vector of maintenance durations
  #Output:
  # The list of elementary moves that creates the full classical neighbourhood

  # init counter
  counter <- 1

  # init moves matrix
  moves <- matrix(NA, ncol=2, nrow=1)

  for (i in 1:n){
    for (j in (max(1,E[i]-wext):min(52-duration[i]+1,L[i]+wext))){
      # add to moves
      moves <- rbind(moves, c(i,j))
      # update counter
      counter <- counter+1
    }
  }

  # return moves without first row
  return(moves[-1,])
}

### function that calculates the objective value of a solution
calcObjVal <- function(x, data, D, M){
  # Input:
  # x = possible solution
  # data = all data
  # Output:
  # objective function value -> sum(rj)^2

  g <- data$G
  Y <- trdata(x = x, data = data, M=M)$Y

  # sum of Pljj's
```

```
  totsum <- 0

  for(j in 1:ncol(Y)){
    # second term
    t2 <- D[j]*(1+S)

    # first term
    t1sum <- 0
    for (i in 1:length(x)){
      gij <- g[i]
      yij <- Y[i,j]
      t1sum <- t1sum + gij*(1-yij)
    }

    # rj
    rj <- max(0, t1sum - t2)

    # rj^2
    val <- rj^2

    # add to total sum
    totsum <- totsum + val
  }

  # return total sum
  return(totsum)
}

### function that generates a random solution
generateRandomSolution <- function(data, wext, M,
                                   W, D, Mjs, K){
  # Input: The problem's full dataset
  # Output: A random solution vector and it's objective function value

  # init vector of solutions
  x <- c()

  for (i in 1:nrow(data)){
    possibleTimes <- c(max(data$E[i] + wext,0):min(52-data$duration[i]+1,
                                                    data$L[i] + wext))
```

```
    # sample from possible staring times
    xi <- sample(possibleTimes, 1)
    # update vector
    x <- c(x, xi)
  }

  # calculate penalty
  P <- checkFeasibilityAndCalculatePenalty(x=x, data=data, M=M, W=W, D=D,
                                          Mjs = Mjs, KK = K)
  # calculate objective value of x, and add P
  objval <- calcObjVal(x = x, data = data, M=M, D = D)
  objval <- objval + P

  # return solution and its oenalised obj val
  return(list(x, objval))
}

### function that transforms x based on ejection chain list
applyChain <- function(x, Chain){\
  # Input:
  # x = previous solution
  # Chain = output of createEjectionChainList
  # Output:
  # transformed solution x
  newx <- x

  for(i in 1:nrow(Chain)){
    # extract unit, time to be changed
    unit <- Chain[i,1]
    time <- Chain[i,2]

    newx[unit] <- time
  }
  return(newx)
}

### function that generates a random solution
initialTemperature <- function(x, xObj, data, M, D,
                               chi0, W, Mjs, K, rwlength=20){
  # Input: The initial solution vector,
```

```
# xObj =  the initial objective function value,
# data = the problem's full dataset
# rwlength = length of random walk
# Output: Two initial temperatures
# calculated using the average increase method
# using the standard deviation method
current <- x
currentObj <- xObj

# init storage of increases and values
increases <- c()
values <- c()

j <- 0

for(i in 1:rwlength){
  # store prev obj function value
  prevObj <- currentObj
  # randomly select a unit
  unit <- sample(1:length(x),1)
  # gen an ejection chain for unit
  chain <- createEjectionChainList(Unit = unit, E = data$E, L = data$L,
                                   x = current, wext = 2, duration = data$duratio
  # apply chain to current x
  # reset current x
  current <- applyChain(x = current, Chain = chain)
  # calculate penalty of current x
  P <- checkFeasibilityAndCalculatePenalty(x = current, data = data, D = D,
                                           M=M, W=W, Mjs = Mjs, KK=K)
  # calculate opbj function value of current x
  currentObj <- calcObjVal(x = current, data = data, D = D, M = M) + P
  # calculate change in obj function from last run
  DeltaE <- currentObj - prevObj
  # if solution got worse
  if(DeltaE > 0){
    # update j
    j <- j+1
    # store increase
    increases <- c(increases, DeltaE)
  }
```

```
    # store obj value
    values <- c(values, currentObj)
  }

  # ave increase in temperature
  avgIncTemperature <- -mean(increases)/log(chi0)
  # sd of objective function
  stdDevTemperature <- sd(values)

  # return two temp options
  return(list(av = avgIncTemperature, sd =stdDevTemperature))
}
################################################
##### GMS local search heuristic ######
runSearchHeur <- function(incumbent, incumbentObj,
                          data, M, D, W, Mj, K){
  # Input:
  # xinc = The incumbent solution vector,
  # objinc = the incumbent objective function value,
  # data = the problem's full dataset
  #Output:
  # The possibly improved incumbent solution vector
  # and corresponding objective function value

  # set incumbent as current
  current <- incumbent
  currentObj <- incumbentObj

  # set improved indicator
  improved <- TRUE

  # create neighbourhood list of current solution
  moves <-createClassicalNeighbourhoodList(n = length(current),
                                           E = data$E, L=data$L,
                                           wext = 2, duration = data$duration)

  while(improved == T){
    # init besst neighbour storage
    bestNeighbour <- c()
    # set arbitrarily large obj function value
```

```
    bestNeighbourObj <- 10^20

    for (i in 1:nrow(moves)){
      neighbour <- current
      # rows to choose from (corresponding to unit i)
      row <- matrix(moves[i,], nrow=1)
      # apply move to neighbour to create new neighbour
      neighbour <- applyChain(x = neighbour, Chain = row)
      # calculate penalty of new neighbour
      P <- checkFeasibilityAndCalculatePenalty(x = neighbour, data = data,
                                               W = W, D = D, Mjs = Mj, KK = K, M =
      # calculate obj value of new neighbour
      neighbourObj <- calcObjVal(x = neighbour, data = data, D = D, M = M)
      neighbourObj <- neighbourObj + P

      # if new neighbour's obj value better than best so far
      if(neighbourObj < bestNeighbourObj){
        # set new neighbour as best neighbour
        bestNeighbour <- neighbour
        bestNeighbourObj <- neighbourObj
      }
    }

    # if final best neighbour is better than current incumbent solution
    if(bestNeighbourObj < incumbentObj){
      # set best neighbour as new incumbent solution
      incumbent <- bestNeighbour
      incumbentObj <- bestNeighbourObj
    }else{ # if best neighbour is not better than current incumbent solution
      # incumbent solution is not improved by any neighbours generated
      improved <- FALSE
    }
  }

  return(list(sol=incumbent, obj=incumbentObj))
}

# function to generate a better random solution
generateGoodRandomSolution <- function(no, data, M, W,
                                       D, Mjs, K){
```

```
  # Input:
  # no = The number of solutions to compare,
  # data = the problem's full dataset
  # Output:
  # A good random solution vector, the objective function value

  # init best obj value as arbitrarily large number
  bestObj <- 10^20

  for (i in 1:no){
    # generate a random solution and obj value
    rand <- generateRandomSolution(data = data, wext = 2, M = M, W = W,
                                   D = D, Mjs = Mjs, K = K)
    solution <- rand[[1]]
    solutionObj <- rand[[2]]
    # apply local search heuristic to improve random solution
    heur <- runSearchHeur(data = data, M = M, D = D, W = W, Mj = Mjs,
                          K = K, incumbent = solution, incumbentObj = solutionObj)
    solution <- heur$sol
    solutionObj <- heur$obj

    # if new solution is better than best so far
    if(solutionObj < bestObj){
      # store solution and its obj value
      best <- solution
      bestObj <- solutionObj
    }
  }

  # return the best solutions from loop
  return(list(sol=best, obj=bestObj))
}

#### HYBRIDISATION ###
runHybrid <- function(data, M, W, D, Mj, K, S, delta,
                      Tmin, omega_fr, initTemp, initSol, initObj){
  #Input:
    # A power system scenario for which to solve the generator
    # maintenance scheduling problem
    # initTemp, initSol previously generated
```

```
#Output:
  # The best maintenance schedule found


# set seed
set.seed(2020)

# init storage of

# starting solution and obj value
currentObj <- initObj

# initial temps using both methods
inittemp <- initTemp
avgT0 <- inittemp[1]
sdT0 <- inittemp[2]

# use avgT0
# temp=T
Temp <- avgT0

# init incumbent solution
incumbent <- current
incumbentObj <- currentObj

# init count of solutions not accepted
notAcceptCounter <- 0

# init vector of all obj function values
obj_all <- c()
obj_allInc <- c()

# init vector of storage of temps
Temps <- c()

# while termination criteria are not met
while((Temp > Tmin) & (notAcceptCounter < omega_fr)){
  # set count of no. accepted, attempted
  numberAccept <- 0
  numberAttempt <- 0
```

```
# init accepted indicator
accepted <- FALSE

while((numberAccept < 12*nrow(data)) & (numberAttempt < 100*nrow(data))){
  # update attempt count
  numberAttempt <- numberAttempt + 1
  print(numberAttempt)
  # init neighbour
  neighbour <- current
  # randomly select a unit
  unit <- sample(1:nrow(data), 1)
  # create ejection chain for neighboour starting at this unit
  chain <- createEjectionChainList(Unit = unit, E = data$E, L = data$L,
                                   wext = 2, x = neighbour, duration = data$dur
  # apply chain on neighbour to get new neighbour
  neighbour <- applyChain(x = neighbour, Chain = chain)
  # calculate feasibility penalty of new neighbour
  P <- checkFeasibilityAndCalculatePenalty(x = neighbour, data = data, W = W, D
                                           Mjs = Mj, KK = K, M = M)
  # calculate objective function value of new neighbour
  neighbourObj <- calcObjVal(x = neighbour, data = data, D = D, M = M)
  # add feasibility penalty
  neighbourObj <- neighbourObj + P
  # calculate change in objective value between current and neighbour
  DeltaE <- neighbourObj - currentObj

  # if neighbour is better
  if(DeltaE <= 0){
    # store neighbour and its objective value
    current <- neighbour
    currentObj <- neighbourObj
    # store obj function value
    obj_all <- c(obj_all, currentObj)

    # update # accepted
    numberAccept <- numberAccept + 1
    # set accepted indicator
    accepted <- TRUE

    # if current obj value is better that incumbent's
```

```
    if(currentObj < incumbentObj){ # line 26
      # replace incumbent solution and obj value
      incumbent <- current
      incumbentObj <- currentObj
      # apply search heuristic to incumbent solution
      heur <- runSearchHeur(incumbent = incumbent, incumbentObj = incumbentObj,
                            data = data, M = M, D = D, W = W, Mj = Mj, K = K)
      incumbent <- heur$sol
      incumbentObj <- heur$obj
      obj_allInc <- c(obj_allInc, incumbentObj)
    }
  }else{ # if neighbour is worse than current
    # if acceptance conditions are met
    if(runif(1) < exp(-DeltaE/Temp)){
      # accept neighboour
      # set current sol and obj value
      current <- neighbour
      currentObj <- neighbourObj
      # store obj function value
      obj_all <- c(obj_all, currentObj)

      # update # accepted
      numberAccept <- numberAccept + 1
      #print(numberAccept)
      # update accepted counter
      accepted <- TRUE
    }
  }
}

if(accepted == TRUE){
  notAcceptCounter <- 0
}else{
  notAcceptCounter <- notAcceptCounter + 1
}

# update temperature - geometric updating
Temp <- GupdateT(Temp = Temp)
#Temp <- VLupdateT(Temp = Temp, Z = obj_all, g = delta)
```

```
    # add to storage
    Temps <- c(Temps, Temp)
  }

  return(list(obj=obj_all, sol=current, inc=incumbent, incObj = incumbentObj,
              incObj_all=obj_allInc, Temps=Temps))
}

### function to check feasibility
is.feasible <- function(x, data, W, D, Mjs, KK, M){
  P <- checkFeasibilityAndCalculatePenalty(x = x, data = data, W = W,
                                    D = D, Mjs = Mjs, KK = KK, M = M)
  # check if feasible
  feas <- P==0

  return(feas)
}

#####################################
# get temps and good solutions beforehand
# 32-unit
S <- S32
TEMPS32 <- c()
INITS32 <- c()
OBJS32 <- c()
# generate init conditions
set.seed(2020)
for (i in 1:50){
  # init sol
  soln <- generateGoodRandomSolution(no = 2, data = data32,
                                   M = M32, W = W32, D = D32, Mjs=Mj32, K=KK32)
  initi <- matrix(soln$sol, nrow=1)
  initiObj <- soln$obj
  # store
  INITS32 <- rbind(INITS32, initi)
  OBJS32 <- c(OBJS32, initiObj)

  # init temp
  tmp <- initialTemperature(x=initi, xObj = soln$obj, data = data32, M = M32,
                           D = D32, chi0 = 0.5, W = W32, Mjs = Mj32, K = KK32,
```

```
                              rwlength = 100)
  # store
  bothTemps <- tmp
  TEMPS32 <- rbind(TEMPS32, matrix(unlist(bothTemps), nrow=1))
}
# save.image("temps32FINAL.RData")

# 21-unit
S <- S21
TEMPS21 <- c()
INITS21 <- c()
OBJS21 <- c()
# generate init conditions
set.seed(2020)
for (i in 1:50){
  # init sol
  soln <- generateGoodRandomSolution(no = 2, data = data21,
                                     M = M21, W = W21, D = D21, Mjs=Mj21, K=KK21)
  initi <- matrix(soln$sol, nrow=1)
  initiObj <- soln$obj
  # store
  INITS21 <- rbind(INITS21, initi)
  OBJS21 <- c(OBJS21, initiObj)

  # init temp
  tmp <- initialTemperature(x=initi, xObj = soln$obj, data = data21, M = M21,
                              D = D21, chi0 = 0.5, W = W21, Mjs = Mj21, K = KK21,
                              rwlength = 100)
  # store
  bothTemps <- tmp
  TEMPS21 <- rbind(TEMPS21, matrix(unlist(bothTemps), nrow=1))
}
library(beepr)
beep()
# save
# save.image("allinitFINAL.Rdata")
# load(file = "allinitFINAL.Rdata")

####### 32-unit run ##########
# set up parallel
```

```
library(doParallel)
cl <- makeCluster(max(1,detectCores() - 1))
registerDoParallel(cl)

########## LOOP 32 ################
# storage of all results
set.seed(2020)
S <- S32

RUNS32 <-foreach(R = 1:3) %dopar% {
        print(paste("run" , R-1, "complete", sep=" "))
        runHybrid(data = data32, M = M32, W = W32, D = D32,
                   Mj = Mj32, K = KK32, S = S32, delta = 0.35,
                  Tmin = 1, omega_fr = 100, initTemp = TEMPS32[R,],
                  initSol = INITS32[R,], initObj = OBJS32[R])
}
beep()
stopCluster(cl)
# save.image(file = "32RunFINALDAY.RData")

# check feasibility
F32 <- c()
for (i in 1:length(RUNS32)){
  F32 <- c(F32, is.feasible(RUNS32[[i]]$sol, data = data32, W = W32, D = D32,
              Mjs = Mj32, KK = KK32, M = M32))
}
F32 # feasible!

##################################################
####################################
####### 21-unit run ##########
cl <- makeCluster(max(1,detectCores() - 1))
registerDoParallel(cl)

########## LOOP 21-unit ################
# storage of all results
set.seed(2020)
S <- S21

RUNS21 <-foreach(R = 1:3) %dopar% {
```

```
  print(paste("run" , R-1, "complete", sep=" "))
  runHybrid(data = data21, M = M21, W = W21, D = D21,
            Mj = Mj21, K = KK21, S = S21, delta = 0.35,
            Tmin = 1, omega_fr = 100, initTemp = TEMPS21[R,],
            initSol = INITS21[R,], initObj = OBJS21[R])
}
beep()
stopCluster(cl)

# check feasibility
F21 <- c()
for (i in 1:length(RUNS21)){
  F21 <- c(F21, is.feasible(RUNS21[[i]]$sol, data = data21, W = W21, D = D21,
                            Mjs = Mj21, KK = KK21, M = M21))
}
F21 # feasible!
#save.image(file = "ALLRUNSFINAL.RData")
#load(file = "ALLRUNSFINAL.RData")
##################################################
##### PLOT RESULTS ######
ALLRUNS <- list(RUNS32, RUNS21)
names(ALLRUNS) <- c("32", "21")

#######
for(test in 1:2){
  pdf(paste("plots", names(ALLRUNS), ".pdf", sep = "")[test],
      width = 15, height = 15, compress = F)
  # set up plot matrix
  par(mfrow=c(3,3))
  # 32 or 21
  runs <- ALLRUNS[[test]]
  for (i in 1:length(runs)){
    # choose run
    thisrun <- runs[[i]]
    ## PLOT
    # obj value
    plot(thisrun$obj, type = "l", ylim = c(0, 4e+07),
           #ylim=c(0, 1e+08), for 21-unit
         lwd = 1, col="cadetblue4",
         ylab="Candidate solution evaluation function value")
```

```
    # inc obj value
    plot(thisrun$incObj_all, type = "l",
         lwd = 2, col = "cadetblue3",
         ylab="Incumbent solution evaluation function value")
    # temp
    plot(thisrun$Temps, type="l",
         lwd = 2, col = "cadetblue2", ylab="Temperature")
  }
  # save pdf
  #dev.off()
}

######## SUMMARY TABLES ########
## 32 UNIT
restable32 <- matrix(NA, ncol = 34, nrow=6)
# first row for names
restable32[,1] <- rep(c("Cand.", "Inc."),3)

for(i in 1:3){
  # solutions
  restable32[2*i-1,2:33] <- RUNS32[[i]]$sol
  restable32[2*i, 2:33] <- RUNS32[[i]]$inc

  # evaluation function values
  restable32[2*i-1,34] <- calcObjVal(x = RUNS32[[i]]$sol,
                                     data = data32, D = D32, M = M32)
  restable32[2*i,34] <- RUNS32[[i]]$incObj
}

#library(xtable)
#xtable(t(restable32))

## 21 Unit
restable21 <- matrix(NA, ncol =23, nrow=6)
# first row for names
restable21[,1] <- rep(c("Cand.", "Inc."),3)

for(i in 1:3){
  # solutions
  restable21[2*i-1,2:22] <- RUNS21[[i]]$sol
```

```
  restable21[2*i, 2:22] <- RUNS21[[i]]$inc

  # evaluation function values
  restable21[2*i-1,23] <- calcObjVal(x = RUNS21[[i]]$sol,
                                     data = data21, D = D21,
                                     M = M21)
  restable21[2*i,23] <- calcObjVal(x = RUNS21[[i]]$inc,
                                   data = data21, D = D21, M = M21)
}

xtable(t(restable21),
       caption = "Results from three runs of 21-unit test system",
       digits = 0)
```