Optimization Project

Implementation of Simulated Annealing for Generator Maintenance Schedules (GMS)

Zach Wolpe

wlpzac001

20 September 2020

Assignment Breakdown

Here we re-implement the application of Simulated Annealing applied to the optimization of 21- & 32-unit generator maintenance schedule (GMS).

The assignment requires us to implement the best/optimal configuration, which, for the $21 - unit\ case$ is given by:

$$\begin{aligned} case &: 21 - unit\ optimal\ implmentation \\ method\ for\ T_0 &: AIM \\ search\ algorithm &: Ejection\ Chain \\ cooling\ scheme &: Van\ Laarhoven \\ \delta &: 0.16 \\ max\ attempts &: 100n \end{aligned}$$

Similarly, the optimal hyper-parameter configuration for the $32 - unit\ caes$ is given by:

$$\begin{aligned} case &: 32 - unit\ optimal\ implmentation \\ method\ for\ T_0 &: SDM \\ search\ algorithm &: Ejection\ Chain \\ cooling\ scheme &: Van\ Laarhoven \\ \delta &: 0.35 \\ max\ attempts &: 90n \end{aligned}$$

We will also examine a number of other configurations, as well as our own slight modification in attempt to achieve better results, & the summary statistics that quantify each of these methodologies & their findings.

Mathematical Representation

The power system consists of n generating units & m time steps. Let $\mathcal{I} = \{1, ..., n\}$ index the set of generating units & $\mathcal{J} = \{1, ..., m\}$ index the set of time periods in the planning horizon. Thus the binary decision variable is given by:

$$x_{ij} = \begin{matrix} 1 \ \ if\ maintenance\ of\ unit\ i\ is\ \textbf{commenses}\ at\ time\ j \\ 0\ otherwise \end{matrix}$$

& auxiliary variable:

$$y_{ij} = \begin{matrix} 1 \ \ if\ unit\ i\ is\ \textbf{under maintenance}\ at\ time\ j \\ 0\ otherwise \end{matrix}$$

Constraints

Let e_i & l_i denote the earliest & latest time periods that maintenance of generating unit $i \in \mathcal{I}$ may start.

Maintenance Window Constraint

$$\sum_{j=e_i}^{l_i} x_{ij} = 1 \ \ i \in \mathcal{I}$$

The model is implemented such that proposed solutions can never be outside of the maintenance window & as such this constraint will always equate to zero, so is merely included for mathematical completeness.

It is known that an item will not be in maintenance outside of the maintenance window, therefore:

$$x_{ij} = 0, \ \ j < e_i \ or \ j > l_i, \ i \in \mathcal{I}$$

$$y_{ij} = 0, \ \ j < e_i \ or \ j > l_i + d_i - 1, \ i \in \mathcal{I}$$

As such we can define the maintenance duration constraint as:

$$\sum_{j=e_i}^{l_i+d-1} y_{i,j} = d_i, \ \ \ i \in \mathcal{I}$$

where d_i is the duration to fix unit i.

Since maintenance is performed over a consecutive periods, we can also add the non-stop maintenance constraint:

$$\begin{matrix} y_{i,j} - y_{i,j-1} \leq x_{i,j}, \ \ i \in \mathcal{I}, \ \ j \in \mathcal{J} \\ y_{i,1} \leq x_{i,1}, \ \ i \in \mathcal{I} \end{matrix}$$

Load Demand Constraint

- $g_{i,j}$: power generation capacity of unit i at time j
- D_j: power demand of time j
- S: safety margin (proportion of D_j)
- r_j: reserve level

Then to ensure the load demand is met we can formulate the constraint:

$$\sum_{i=1}^{n} g_{i,j}(1 - y_{i,j}) = D_j(1 + S) + r_j, \quad j \in \mathcal{J}$$

which intuitively reads, the generators in use should produce the equivalents of the demand, a safety margin & a reserve - thus the reserve that we ought to minimize.

Maintenance Crew Constraint

We are bound by the availability of power maintenance staff. Let $m`_{p,ij}$ denote manpower for unit i when in maintenance during time period j - if maintenance of this unit were to commence in time period p. If m_i^k denotes the required manpower of unit i in the k_{th} period of maintenance.

$$\begin{aligned} m'_{p,ij} &= m_i^{j-p+1} \quad j - p < d_i \\ &= 0 \quad otherwise \end{aligned}$$

The maintenance crew constraint can be given by:

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

Where M_j denotes the available manpower in time period $j \in \mathcal{J}$.

Exclusion Constraint

Certain units cannot be maintained simultaneously. $\mathcal{K}$ denotes the set of indices of generating unit exclusion subsets $\mathcal{K} = \{1, ..., K\}$,

If K_k denotes the maximum number of units within subset $\mathcal{I}_k$ ($\mathcal{I}_k$) formalized as:

$$\sum_{i \in I_k} y_{i,j} \leq K_k$$

Note that no exclusion set (thus no exclusion constraint) is given for the 21 unit case.

Variable Definition Constraints

finally we constrain the nature of the variables:

$$\begin{aligned} x_{i,j} \in \{0, 1\}, \;\; i \in \mathcal{I}, \;\; j \in \mathcal{J} \\ y_{i,j} \in \{0, 1\}, \;\; i \in \mathcal{I}, \;\; j \in \mathcal{J} \\ r_j \geq 0, \;\; j \in \mathcal{J} \end{aligned}$$

Objective Function

The nature objective is then to minimize the surplus power produced - reserve load:

$$\underset{r}{argmin} \sum_{j=1}^{m} (D_j S + r_j)^2$$

This yields a *mixed integer quadratic program formulation of the GMS problem.* This is subject to:

$$r_j \geq 0 \quad \forall j \in \mathcal{J}$$

Where r_j is computed as the excess power generated after demand & the safety margin is satisfied.

Solution Representation

A candidate solution is represented by:

$$vector\ \mathbf{x} = \{x_1, ..., x_n\}$$

Where x_i takes on *integer values* that indicate which period unit i's maintenance commences.

Given the solution vector & the data, all other objects can be computed.

Weighted Penality

In the event that any constraints are violated we compute a penalty which is added to the objective function.

$$P = w_w P_w + w_l P_l + w_c P_c + w_e P_e$$

Soft Constrain Violation Approach

A soft constraint approach is applied to this implementation - meaning we accept infeasible solutions. This provides an important advanced as the solution space can be search for better solutions whilst still technically in an infeasible region.

A *hard constraint* search would reject all solutions that do not fully satisfy the practicalities. If the problem is heavily constrained it may take arbitrarily long to find an initial solution that is accepted, thus the majority of the search time is unintelligent brute force random search/grid search. This would be highly inefficient & may not find a solution at all.

This *soft constraint* approach instead accepts poor solutions but penalizes the objective function, to encourage search to move towards feasible solutions rapidly.

The *penalty* is computed as a weighted sum of the constraint violations. Larger weights discourage exploration, small weights increase the probability of the incumbent solution being infeasible. This trade-off needs

be balanced.

Soft Constraint Approach Details

Larger violations receive larger penalties to discourage infeasible candidate solutions. Each constraint violation is computed as:

Maintenance Window Constraint Penalty: the number of periods the maintenance occurs before the e_i or after l_i:

$$P_w^i = \begin{cases} e_i - x_i, & \text{if } x_i < e_i \\ 0, & e_i < x_i < l_i \\ x_i - l_i, & l_i < x_i \end{cases}$$

Which is computed for each unit, thus the total weighted penality is given by:

$$P_w = \sum_{i=1}^{n} w_w^i P_w^i$$

Load Demand Constraint Simple given as the shortfall of load in the period:

$$P_l^j = max\{-r_j, 0\} \quad \forall j \in \mathcal{J}$$

Thus:

$$P_l = \sum_{j=1}^{m} P_l^j$$

Crew Manpower Constraint P_c^j gives the shortfall of manpower during time period j.

$$P_c^j = max\ \{\sum_{i=1}^{n}\sum_{p=1}^{j} m'_{p,i,j} x_{i,p} - M_j, 0\}$$

Thus the overall crew manpower penalty is the sum over all time periods (j):

$$P_c = \sum_{j=1}^{m} P_c^j$$

Exclusion Constraint

$$P_e^{k,j} = max\{\sum_{i \in I_k} y_{i,j} - K_k, 0\}$$

Thus the aggregate is given by:

$$P_e = \sum_{k=1}^{K} \sum_{j=1}^{m} P_e^{k,j}$$

Note: that some electrical generators are not given exclusion groups, thus can be maintained concurrently with any other units & yield no penalty.

Total Penalty The total penalty is then given as:

$$P = w_w P_w + w_l P_l + w_c P_c + w_e P_e$$

Data

Now that we have a model formulation, we need to utilize data from the literature to parameterize our model. Here is the data available for both the 21 & 32 unit cases.

The function below is used to load the datasets, either 21 or 32 unit datasets & can be called at any point to change the dataset.

Note that the matrix *el* needs to be computed outside of the function as we cannot cbind() to update global parameters within a function - all parameters are updated globally when calling *load_data*. Additionally not the binary parameter *IEEE_RTS_system* is used to select 21 or 32 unit caes: if *IEEE_RTS_system=True* we are dealing with the data of the 32 unit case.

```
# ---- Simulated Annealing ----x

# ---- Parameters ----x
# n units:                (21 or 32) units
# hyper parameters:       i gi ei ‘i di mki i gi ei
# exclusion set:          (i in Ik) Kk
# weekly load demand:     j Dj

# ------------------------------------------------ Data  ----------------------------------------
load_data <- function(IEEE_RTS_system=TRUE) {
  if (IEEE_RTS_system) {
    # ------------------------------------------- 32-Unit  ---------------------------------------

    # ----- Hyper-parameters -----x
    size <<- 32
    n <<- 32
    m <<- 52

    # exclusion sets data
    excl_set <<- c(rep(1,4), rep(2,4), rep(3,3), rep(4,3), rep(5,6), rep(NA,3), rep(6,6), rep(7,3))
```

```
  K <<- cbind(unique(excl_set)[!is.na(unique(excl_set))], c(2,2,1,1,3,3,1))

  # weekly load demand
  demand <<- c(2457, 2565, 2502, 2377, 2508, 2397, 2371, 2297, 2109, 2100, 2038, 2072, 2006,
               2138, 2055, 2280, 2149, 2385, 2480, 2508, 2440, 2311, 2565, 2528, 2554, 2454,
               2152, 2326, 2283, 2508, 2058, 2212, 2280, 2078, 2069, 2009, 2223, 1981, 2063,
               2063, 2118, 2120, 2280, 2511, 2522, 2591, 2679, 2537, 2685, 2765, 2850, 2713)
  S <<- 0.15

  # hyper parameters
  g <<- c(20, 20, 76, 76, 20, 20, 76, 76, 100, 100, 100, 197, 197, 197, 12, 12, 12, 12, 12, 155, 155,
  e <<- c(1, 1, 1, 27, 1, 27, 1, 27, rep(1, 5), 27, rep(1,6), 27, 1, 27, rep(1,9))
  l <<- c(25, 25, 24, 50, 25, 51, 25, 50, 50, 50, 50, 23, 23, 49, 51, 51, 51, 51, 51, 23, 49, 21, 47,
  d <<- c(2,2, 3,3, 2,2, rep(3,5), rep(4,3), rep(2,5), 4,4, 6,6, rep(2,6), 4,4, 5)
  mk <<- c()
  mk[[1]] <<- c(7,7); mk[[2]] <<- c(7,7); mk[[3]] <<- c(12,10,10); mk[[4]] <<- c(12,10,10)
  mk[[5]] <<- c(7,7); mk[[6]] <<- c(7,7); mk[[7]] <<- c(12,10,10); mk[[8]] <<- c(12,10,10)

  mk[[9]] <<- c(10,10,15); mk[[10]] <<- c(10,10,15); mk[[11]] <<- c(15,10,10); mk[[12]] <<- c(8,10,10
  mk[[14]] <<- c(8,10,10,8); mk[[15]] <<- c(4,4); mk[[16]] <<- c(4,4); mk[[17]] <<- c(4,4); mk[[18]] <
  mk[[20]] <<- c(5,15,10,10); mk[[21]] <<- c(5,15,10,10); mk[[22]] <<- c(15,10,10,10,10,5); mk[[23]] <
  mk[[24]] <<- c(6,6); mk[[25]] <<- c(6,6); mk[[26]] <<- c(6,6); mk[[27]] <<- c(6,6); mk[[28]] <<- c(6
  mk[[30]] <<- c(12,12,8,8); mk[[31]] <<- c(12,12,8,8); mk[[32]] <<- c(5,10,15,15,5)
  Mj <<- max_main_personal <<- 25

  # ---- 32-Unit Weights ----x
W <<- c(40000, 1, 20000, 20000)
}
else {
  # ------------------------------------------------ 21-Unit  ----------------------------------------

  # ----- Hyper-parameters -----x
  size <<- 21
  n <<- 21
  m <<- 52

  # exclusion sets
  # no exclusion

  # load demand
  demand <<- rep(4739,52)
  S <<- 0.15

  # hyper parameters
  g <<- c(rep(555,2),rep(180,2),rep(640,3),555,276,140,90,76,76,94,39,188,58,48,137,469,52)
  e <<- c(1,27,1,1,27,1,1,27,1,1,1,27,1,1,1,1,27,27,27,27,1)
  l <<- c(20,48,25,26,48,24,24,47,17,23,26,50,25,23,25,25,52,51,52,49,24)
  d <<- c(7,5,2,1,5,3,3,6,10,4,1,3,2,4,2,2,1,2,1,4,3)
  mk <<- c()
  mk[[1]] <<- c(10,10,5,5,5,5,3); mk[[2]] <<- c(10,10,10,5,5); mk[[3]] <<- c(15,15); mk[[4]] <<- c(20
  mk[[6]] <<- c(15,15,15); mk[[7]] <<- c(15,15,15); mk[[8]] <<- c(10,10,10,5,5,5); mk[[9]] <<- c(3,rep
  mk[[11]] <<- c(20); mk[[11]] <<- c(20); mk[[12]] <<- c(10,15,15); mk[[13]] <<- c(15,15); mk[[14]] <
  mk[[16]] <<- c(15,15); mk[[17]] <<- c(20); mk[[18]] <<- c(15,15); mk[[19]] <<- c(15); mk[[20]] <<- c
```

```
    Mj <<- max_main_personal <<- 20

    # ---- 21-Unit Weights ----x
    W <<- c(500000, 1, 200000, 0)
  }
}

load_data(IEEE_RTS_system = TRUE)
el <- cbind(e,l)
```

Binary Decision Variables: Y

In order to perform all the subsequent computations, we need to compute the y binary matrix that encodes information about when certain units are under maintenance.

The x solution matrix details the commencement period - that is, when unit i started it's maintenance period. This does not, however, detail any information as to how long the unit is under maintenance.

To aid subsequent calculations we compute y, a binary matrix detailing when unit i is under maintenance (period j) - computed by the data, which gives a maintenance duration of each.

Each row of y represents a unit & each column a time period j, thus the matrix is sparse (mostly 0) & is 1 where maintenance is conducted for unit i in period j - deterministically calculated from the starting periods - given by the solution encoding x - & the maintenance duration period of unit i - given by the data.

The function below computes this y matrix, given the data & current (global) solution x.

```
# ---- compute binary decision variable ----x

# ---- compute x ----x
# init


# ---- compute y ----x
# x=x, size=size, m=m
compute_y_binary_matrix <- function() {
  y <- as.data.frame(matrix(0, nrow=size, ncol=m))
  names(y) <- 1:m

  update_y_rows <- function(ind) {
    if ((x[ind]+d[ind]-1) <= m) y[ind, x[ind]:(x[ind]+d[ind]-1)] <<- 1
    else {
      # wrap around
      excess_len <- (x[ind]+d[ind]-1) - m
      y[ind, x[ind]:m] <<- 1
      y[ind, 1:excess_len] <<- 1
    }
  }

  # update y
  invisible(lapply(1:length(x), update_y_rows))
```

```
  return(y)

}
```

Penalty Weights

Before the penalties can be computed we need to specify the penalty weights. The paper runs a series of experiments to select the weights, coming to the final results:

21-Unit System The penalty weights are given by:

$Load\ Demand\ Constraint = 1$

$Load\ Demand\ Constraint = 1$

$$Maintenance\ Window\ Penalty = 500'000$$
$$Load\ Demand\ Constraint = 1$$
$$Maintenance\ Crew\ Constraint = 200'000$$

These hyper-parameters where chosen to achieved an optimal level of feasible solution. The 21-unit case does not have an exclusion constraint.

32-Unit System Similar to the aforementioned system, the 32-unit option was chosen such to converge on accepting a suitable number of proposed feasible solutions.

$$Maintenance\ Window\ Penalty = 40'000$$
$$Load\ Demand\ Constraint = 1$$
$$Maintenance\ Crew\ Constraint = 20'000$$
$$Exclusion\ Constraint = 20'000$$

For brevity, the weights are updated in the *load data* function.

Generate Initial Random Solution

We initialize the algorithm by defining a initial random solution, in which all maintenance points in $\mathbf{x} = \{x_1, x_2, ..., x_I\}$ are chosen randomly from a uniform distribution between their respective earliest e_i & latest periods l_i.

The function can be used for both the 21 & 32 unit case, the parameter *ii* is used for efficient computation (allows one to use *apply* & *lapply* later on) & serves no purpose.

```r
# ------------------------------------------------ Initialization  ----------------------------------
# I: 1-32        size <- 32
# J: 1-52 (m)    m <- 52
init_solution <- function(ii=F, size=size, m=m, e=e, l=l) {
  unlist(lapply(1:size, function(ind) sample(el[ind,1]:el[ind,2], size = 1)))
}
```

Calculations: Penalties + Objective Function

Now, given a solution, we compute the penalty incurred by violating any constraints.

Each Penalty is computed individually & thereafter weighted by it's respective weight & aggregated to compute the total penalty.

Maintenance Window Constraint The algorithm is defined such that proposed solutions can only be made during the feasible window, so it is impossible for this constraint to exceed 0, however we include it so that different specifications of the algorithm can be computed if so desired.

For each element i we compute:

$$P_w^i = max\{e_i - x_i, 0\} + max\{x_i - l_i, 0\}$$

Thus if x_i is within the range $e_i \leq x_i \leq l_i$ no penalty is incurred & a penalty is incurred if x_i lies outside of the range. Specifying both in terms of a *max* as aposed to an equivalent *min* also allows one to compute the penalty directly.

We try to run as many computations in parallel as possible (thus we frequently use the *apply* family) as it speeds computation.

Load Demand Constraint The code may look a bit cryptic as it is performed in few lines, however the operations are simple. First, given the generating capacity of each unit g_i we need to compute generating capacity for *each time step*, that is for all available generators.

The y matrix - which gives the binary indicator of whether or not unit i is under maintenance during period j, & as of dimensions $n \times m$ - is already computed for the candidate solution in mention. As such we use the y matrix to select all elements where $y = 0$ - that is all units *NOT* under maintenance - & compute their combined generating capacity for *EACH PERIOD*. This is given by the variable *g_pwr*.

S gives the proportion of electricity above the demand needed as a safety, thus we actually need *demand* $\times$ $(1 + S)$.

We this simple compute the reserve as:

$$r_j = \sum_{i=1}^{n} g_{ij}(1 - y_{ij}) - D_j(1 + S)$$

Thus we simple compute the penalty as:

$$P_l^i = max\{-r_j, 0\}$$

Crew Constraint We first need to compute $m`_{p,ij}$: which we encode as a matrix of size $n \times m$ that gives the maintenance crew *required* for the specific unit whilst under-maintenance (thus a sparse, mostly zero, matrix with the crew requirements given in the locations analogous to the y matrix showing when the unit is under maintenance).

Note that I wrote these function as to *wrap around* if the period exceeds the *52* time periods available. This will not be utilized given the *strong* maintenance window constraint, however it was provided for a more robust solution that is adaptable to other problems.

One $m`_{p,ij}$ is computed, the column sums of $m`_{p,ij}$ give us the maintenance crew required for each time period j. We thus compute the penalty as

$$max\{Colsum\{m`_{p,ij}\} - M_j, 0\}$$

which, if positive, implies that the labour required is greater than the labour available - thus is the penalty.

Exclusion Constraint We need to ensure that elements of the same group do not exceed the specified limit. First notice that:

1. The 21-Unit system does not have an exclusion constraint, & no data is required, thus will always be zero.
2. The 32-Unit system has 3 elements that do not belong to any group. We will treat these as *unbound*, putting no exclusion constraint on them.

The matrix K gives the groups (col 1) & their limits. Note that those without a group are assigned to '0' which is not in the table. We simple use the matrix K to filter for the group of interest & sum each column of y matrix that corresponds to the the group - indicating how many are active simultaneously. This vector (a number for each time step) should not exceed the specified limit, if it does it should generate a penalty only *by the amount it exceeds the penalty* (which is why we subtract the limit.

Aggregate Constraint Once computed individually compute the weighted sum of each penalty to produce the aggregate penalty.

Objective Function

The objective function is also computed in this cell.

We ought to compute the reserve r_j for each time j in order to compute the objective function. Earlier we computed any negative values of r_j to compute the load demand penalty. We computed r_j as:

$$r_j = \sum_{i=1}^{n} g_{ij}(1 - y_{ij}) - D_j(1 + S)$$

& thereafter the the load constraint was calculated as:

$$P_l^i = max\{-r_j, 0\}$$

The model, however, restricts $r_j \geq 0$, & including negative values would decrease the objective function. As such for the purpose of the objective function we compute the reserves as:

$$r_j = max\{\sum_{i=1}^{n} g_{ij}(1 - y_{ij}) - D_j(1 + S), 0\}$$

thus do not allow negative values:

$$r_j \geq 0 \; i \in J$$

Thus negative values are allowed in computing the constraint violation but not when computing reserve, because they are *not* reserve if negative (but rather a deficiency). Including negative numbers would not make sense as it would *decrease* the objective function:

$$\sum_{j=1}^{m} (D_j S + r_j)^2$$

thus suggesting this would be a better solution, which counteracts the penalty & is counter-productive. This also makes sense as reserve is defined as *unused energy* & not the different/surplus/deficit in energy.

Function

The function below computes the penalty & objective function for any configuration, it returns the individual (weighted) penalties as well as the unpenalized & penalized objective functions. It also returns a binary indicator called *feasible* that can be used to assess if a solution incurred a penalty or not - that is, if it is feasible.

```
# ------------------------------------------------   Compute Penalties   ------------------------------
compute_penalty <- function(ii=FALSE, IEEE_RTS=TRUE) {


  # ---------- Maintenance Window Constraint ----------x
  Pw <- matrix(0, nrow = size, ncol = 1)
  update_Pw <- function(ind) {
    Pw[ind] <<- max(e[ind] - x[ind], 0) + max(x[ind] - l[ind], 0)
  }
  lapply(1:size, update_Pw)



  # ---------- Load Demand Constraint ----------x
  # generated power
  g_pwr <- unlist(lapply(1:m, function(ind) sum(g[!y[,ind]])))
  r_j <- g_pwr - demand*(1+S)
  Pl <- r_j
  Pl[r_j > 0] <- 0
  Pl[!r_j > 0] <- -r_j[!r_j > 0]


  # ---------- Maintenance Crew Constraint ----------x
  # compute m_{pij}
  mp <- matrix(0, nrow = size, ncol = m)
  update_Mpij <- function(ind) {
    if ((x[ind]+d[ind]-1) <= 52) {
      mp[ind, x[ind]:(x[ind]+d[ind]-1)] <<- mk[[ind]]
    } else {
      # wrap around
      excess_len <- (x[ind]+d[ind]-1) - 52
```

```
      mp[ind, x[ind]:52] <<- mk[[ind]][c(1:(length(mk[[ind]]) - excess_len))]
      mp[ind, 1:excess_len] <<- mk[[ind]][-c(1:(length(mk[[ind]]) - excess_len))]
    }
  }
  lapply(1:size, update_Mpij)
  Pc2 <- unlist(lapply(1:m, function(i) max(colSums(mp)[i]-Mj, 0)))
  # alterative method
  Pc <- colSums(mp) - Mj
  Pc[Pc <= 0] <- 0



  # ---------- Exclusion Constraint ----------x
  excl_set[is.na(excl_set)] <- 0

  # for group K=1
  Pe <- 0
  for (i in 1:nrow(K)) {
    sub <- excl_set == K[i,1]
    inds <- colSums(y[sub,]) > K[i,2]
    Pe <- Pe + sum(colSums(y[sub,])[inds] - K[i,2])
  }


  # ---------- Total Penalty ----------x
  Pw <- sum(Pw)
  Pl <- sum(Pl)
  Pc <- sum(Pc)
  if (!IEEE_RTS) Pe <- 0

  P <- c(Pw, Pl, Pc, Pe)
  weighted_P <- W * P
  Pw <- weighted_P[1]; Pl <- weighted_P[2]; Pc <- weighted_P[3]; Pe <- weighted_P[4];
  P <- sum(weighted_P, na.rm = T)

  # ---------- Objective Function ----------x
  r_j <- apply(matrix(r_j), 1, function(ii) max(ii, 0))
  Q <- sum((demand*S + r_j)^2)
  Q_total <- Q + P

  # ---------- Feasibility Check ----------x
  feasible <- TRUE
  if (P > 0) feasible <- FALSE

  return(list(P=P, Pw=Pw, Pl=Pl, Pc=Pc, Pe=Pe, Q=Q, Q_total=Q_total, feasible=feasible))
}
```

Neighbourhood Search

Note: that random choices are made in accordance with uniform sampling.

Classical Operator

The traditional perturbation operator utilized in simulated annealing, the classical operator simple changes 1 node at random to explore the solution space. This is done by - given an initial solution:

1. Select 1 unit at random
2. Randomly change it's starting time (within the allowed window) by sampling from the available space in a uniform fashion.

Ejection Chain Operator

A more aggressive operator, that more violently searches the solution space. This operator is said to explore the solution space more effectively by utilizing a chain dependent sequence to capture more global information/structure (about the entire sequence).

The Ejection Chain Operator is performed, given an initial solution x containing maintenance starting times, by:

1. An initial unit is selected at random.
2. It's starting time is randomly changed (within the allowed window).
3. A unit whose starting time is the same as the new (randomly chosen) is now sampled at random.
4. Repeat steps $2-to-3$ until no element matches the newly chosen number *or* the newly select starting time corresponds to the initial starting time of the initial sampled node/unit.

Below we implement both the classical move operator & the ejection chain operator - both functions taking a solution input & return a proposed move.

As used elsewhere, the parameter *ii* is simple a placeholder to call the function many times using the *apply* family & can be ignored.

```
# ------------------------------------------------- Neighbourhood Search --------------------------------


# ---------------------- Classical Search Operator ----------------------x
classical_operator <- function(sol=x, el=el, ii=F) {
  ind <- sample(1:length(sol), 1)
  sol[ind] <- sample(el[ind,1]:el[ind,2], size = 1)
  sol
}




# ---------------------- Ejection Chain Search Operator ----------------------x
ejection_chain_operator <- function(sol=x, el=el, ii=F) {

  # ---- first random selection ----x
  ind <- sample(1:length(sol), 1)
  start_start_time <- sol[ind]
  new_move <- sample(el[ind,1]:el[ind,2], size = 1)
```

```
  # ---- run through chain ----x
  BREAK <- FALSE
  while (new_move != start_start_time & BREAK==FALSE) {

    # ---- perturb new move ----x
    new_move <- sample(el[ind,1]:el[ind,2], size = 1)
    sol[ind] <- new_move

    # ---- check if chain is complete ----x
    if (sum(sol == new_move) <= 1) BREAK <- T # after update
    else {

      # ---- compute new index ----x
      sub <- which(sol == new_move)
      sub <- sub[sub != ind]
      ind <- sample(sub, size = 1)

    }
  }
  return(sol)

}
```

Hybrid Search

Adding complexity provides a constant tradeoff between being able to more effectively search the solution space & incurring additional computational costs.
The paper proposes a hybrid search solution: which adds computational cost but may result in more effect search. The hybrid model adopts the classical search operator & for each move in the cooling process, whenever a new incumbent solution is found, it searches locally for the best possible neighbouring solution within the neighbourhood defined by nm.
Importantly, the local search only updates the incumbent solution - that is the stored best known solution - & does not effect the current solution. It does this to avoid premature convergence, as this would converge to a local optima.
The Hybrid Search combines the current Simulated Annealing Architecture with a local hill-climber (deterministic) search mechanism.
The local search mechanism is performed around the current *incumbent solution* - that is the best known solution. We this periodically perform a local search around the current best known solution, whilst the full algorithm is described in *Algorithm 4.8* on page 73 of the thesis, the algorithm essentially: 1. Given the current incumbent solution 2. Compute n neighbourhood search moves (moves according to the classical operator) 3. Iterate through each proposed move 4. Select the best (lower objective function) perturbation & set this as the incumbent solution.
A final hyper-parameter is to determine how often this local search should be conducted. Two variations of this are offered: the local search could be computed whenever a new incumbent solution is achieved, or at the end of the full runtime.
Local Search Heuristic
The paper also proposes a hybrid adaptation to the model, in which the local search nature of the meta-heuristic are utilized to explore the neighbourhood of the *incumbent solution* more prudently. This adaptation, given below, is easy to implement as it simply searches around the incumbent solution more thoroughly - using the ejection chain local search operator.

Algorithm 4.8: The GMS local search heuristic

Input: The incumbent solution vector, the incumbent objective function value, the problem's full dataset

Output: The possibly improved incumbent solution vector and corresponding objective function value

```
1  [current, currentObj] ← [incumbent, incumbentObj]
2  improved ← true
3  moves ← createClassicalNeighbourhoodList(n, e, ℓ, W_ext)
4  while improved = true do
5      bestNeighbour ← ∅
6      bestNeighbourObj ← some very large number
7      for i ← 1 to number of elements in moves do
8          neighbour ← current
9          Apply moves(i) on neighbour to create new neighbour        // i-th move
10         P ← checkFeasibilityAndCalculatePenalty(neighbour, dataset)
11         Calculate neighbourObj
12         neighbourObj ← neighbourObj + P
13         if neighbourObj < bestNeighbourObj then
14             [bestNeighbour, bestNeighbourObj] ← [neighbour, neighbourObj]
15         end
16     end
17     if bestNeighbourObj < incumbentObj then
18         [incumbent, incumbentObj] ← [bestNeighbour, bestNeighbourObj]
19     else
20         improved ← false
21     end
22 end
23 return [incumbent, incumbentObj]
```

Two variants are proposed:
1. Local search is applied each time the incumbent solution is updated 2. Local search is applied at the end on the final incumbent solution
Only the incumbent solution is updated & not the current solution, to avoid pre-mature convergence, the *track* or *direction* of the algorithm is unaffected by this local search.
The approach searches the full neighbourhood of the incumbent solution, of maximum size nm. Here we implement a function that allows for this local search. The algorithm provided runs until an improvement is found, in our implementation we allow for 2 options: run until an improvement is found, by setting *iters=NA* or to run for a pre-specified number of iterations. This is done by setting the count value to end the *while* loop after a specified number of iterations, if provided.
Note that some of the orders of execution have changed to speed computation, like finding all neighbours in parallel, though the functionality is identical to that provided in the thesis.
"'r local_search_hybrid <- function(incumbent_sol, neighbourhood_size=size*m, iters=1000, search_operator=ejection_chain_operator, prints=TRUE) {
improve <- FALSE; cnt <- 1; found_imp <- F best_neighbour <- incumbent_sol x «- incumbent_sol; y «- compute_y_binary_matrix() best_scores <- inc_score <- compute_penalty()$Q_total while (improve!=TRUE) {
# —- compute neighbourhood_size number of neighbours —-x nh <- lapply(1:neighbourhood_size, ejection_chain_operator, sol = best_neighbour, el = el)

— find best neighbour —-x scores <- c() for (i in nh) { x «- i; y «- compute_y_binary_matrix() scores <- c(scores, compute_penalty()$Q_total) } ind <- which(scores == min(scores))[1] best_neighbour <- nh[[ind]] if (scores[ind] < best_scores) { incumbent_sol <- best_neighbour best_scores <- scores[ind] improve <- TRUE found_imp <- T } # —- count if run length given —-x if (!is.na(iters)) if (cnt==iters) improve <- TRUE # —- count iters —-x cnt <- cnt + 1 } if (prints & found_imp) { print(‘.......... Improvement Found! ....’) print(‘.......... Improvement not found ....’) } if (prints & !found_imp) { print(‘.....’) print(paste(‘Previous Objective:’, inc_score)) print(‘.....’) print(paste(‘New Objective:’, best_scores)) print(‘.....’) } return(list(incumbent_sol=incumbent_sol, score=best_scores)) } x <- init_solution(size=size, m=m, e=e, l=l) local_search_hybrid(incumbent_sol = x, neighbourhood_size = size*m, iters = NA, search_operator = ejection_chain_operator) “‘

```
## [1] ".......... Improvement Found! .........." ## [1] ".......... Improvement not found .........."
## $incumbent_sol ##  [1]  4 20  9 33 16 39 23 35  3 47 15 17  2 47 36 49  7 51 15 21 40 11 39 33 38 ## [26] 11 17 26 41  8 44 29 ## ## $score ## [1] 37414769
```

Improved Initial Solution

Another local search heuristic adaptation is to perform local search on the initial solution - yielding a ‘good’ random solution. The same function as above can be applied to a starting configuration.

This method appears to simple add an additional search that could be considered part of the full execution, with a special case hard acceptance probability. It may also lead to pre-mature convergence & should be used with caution.

Note, however, that the algorithm provided in the paper for this function gives a terminal count (max number of attempts) & does not run *until* (using a *while*) a better candidate is found.

Initial Temperature

In order to initialize our model, we need to estimate the T_0. Two methods for doing so are detailed in the paper: *average increase method* & the *standard deviation method*. The algorithm for determining the two starting temperatures is show here:

Function 4.6: `initialTemperature(`$\boldsymbol{x}, xObj, dataset$`)`

Input: The initial solution vector, the initial objective function value, the problem's full dataset

Output: Two initial temperatures calculated using the average increase method and using the standard deviation method

```
[current, currentObj] ← [x, xObj]
j ← 0
for i ← 1 to length of random walk do
    previousObj ← currentObj
    unit ← rand([1, n])
    chain ← createEjectionChainList(unit, n, e, ℓ, W_ext, current)
    Apply chain on current to create new solution
    P ← checkFeasibilityAndCalculatePenalty(current, dataset)
    Calculate currentObj
    currentObj ← currentObj + P
    ΔE ← currentObj − previousObj
    if ΔE > 0 then
        j ← j + 1
        increases_j ← ΔE
    end
    values_j ← currentObj
end
avgIncTemperature ← −average(increases) / ln(χ_0)
stdDevTemperature ← stdDeviation(values)
return [avgIncTemperature, stdDevTemperature]
```

Average Increase Method: AIM Given an initial solution we need to compute a starting temperature to execute the algorithm. This method relies are determining

$$\mathcal{X}_0 \leftarrow \textit{initial acceptance ratio}$$

$$\bar{\Delta E}^{(+)} \leftarrow \textit{agerage increase in energy}$$

First we need to compute as estimate $\bar{\Delta E}^{(+)}$: Which is estimated by executing a random walk over the solution space, given the random starting configuration, & averaging the increase in temperature. Thereafter we can estimate the starting configuration as:

$$\mathcal{X}_0 = exp(-\frac{\bar{\Delta E}^{(+)}}{T_0})$$

Thus, the initial temperature under *average increase method*:

$$T_0 = -\frac{\bar{\Delta E}^{(+)}}{ln(\mathcal{X}_0)}$$

Note that:

$$\Delta E \leftarrow currentObj - previousObj$$

Keep in mind that lower objective functions are better, as such if $\Delta E < 0$ we have moved to a *better location*. Thus if

$$\Delta \bar{E}^{(+)} := \Delta E > 0$$

denotes *worse moves*.

Further, although $\mathcal{X}_0$ is a free hyper-parameter, the paper notes that in our instance:

$$\mathcal{X}_0 \leftarrow 0.5$$

Standard Deviation Method: SDM We can just as easily implement the standard deviation method, whereby the same random walk is implemented & the standard deviation of all objective functions achieved during the random walk is computed & used to set $T0$, that is:

$$T_0 = std(values) = \sigma_0$$

Compute AIM & SDM Note: the random walk is conducted using the ejection chain method - not the classical operator - as per the algorithm detailed in the thesis. Here we define a function that performs algorithm 4.6:

Note: parallel computation cannot be utilized because successive calculations are dependent on prior iterations. The function below returns both *AIM* & *SDM*.

```
# ---- example init solution ----x
x <- init_solution(size=size, m=m, e=e, l=l)


# ---- find init temperature ----x
compute_init_temperature <- function(x=x, x0=0.5, walk_length=500) {
  # note: the function will modify x directly

  random_walk_solutions <- c()
  # --- compute walk ---x
  for (i in 1:walk_length) {
    x <<- ejection_chain_operator(sol=x, el=el)
    y <<- compute_y_binary_matrix()
    res <<- compute_penalty()
    random_walk_solutions <- c(random_walk_solutions, res$Q)
  }

  # --- compute ave increase ---x
  ave_inc <- c()
  for (i in walk_length:2) {
    delta_E <- random_walk_solutions[i] - random_walk_solutions[i-1]
    if (delta_E > 0) ave_inc <- c(ave_inc, delta_E)
  }
  mean_av_inc <- mean(ave_inc)
```

```
  # --- compute ave temp AIM ---x
  T0 <- -mean_av_inc/log(x0)

  # --- compute sigma_0 ---x
  sd <- sd(random_walk_solutions)

  return(list(ave_inc=T0, sd0=sd))
}


T0 <- compute_init_temperature(x = x, x0 = 0.5, walk_length = 500)
T0_aim <- T0$ave_inc; T0_std <- T0$sd0


print('_____Example: Compute Init Temperture:______')
```

```
## [1] "_____Example: Compute Init Temperture:______"
```

```
print(paste('Average Increase in Temperature: ', round(T0_aim, 2)))
```

```
## [1] "Average Increase in Temperature:  878887.29"
```

```
print(paste('Standard Deviation of Random Walk: ', round(T0_std, 2)))
```

```
## [1] "Standard Deviation of Random Walk:  653969.23"
```

Cooling Schedules

---

4 cooling schedules are utilized. The standard Geometric cooling as well as 3 adaptive cooling schedules. They are adaptive in that they are a function of the current state.

Geometric Cooling

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

Which is the most frequently used/standard cooling schedule for

3 adaptive approaches to cooling are also implemented. They are adaptive as they receive feedback from the algorithm to tune the next temperature, addressing 2 contridictory goals:

- keep the annealing close to equilibrium
- execute the annealing in the shortest possible time

In all instances σ_s is the adaptive parameter.

Huang

$$T_{s+1} = T_s \; exp\{-\frac{\lambda T_s}{\sigma_s}\}$$

Where λ is some control parameter & σ_s, the adaptive parameter, is computed as *the standard deviation observed in the changing values of the objective function when reaching stage s.* That is, the standard deviation of all observed objective values so far.

Van Laarhoven

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

Where δ is _some small real number.

Triki

$$T_{s+1} = T_s(1 - T_s \frac{\Delta}{\sigma_s^2})$$

Where Δ is the *expected decrease in average objective function value* when reaching the next temperature stage of the search process.

Δ is initally defined by (where $\mu_2 \in [1, 20]$:

$$\Delta = \frac{\sigma_0}{\mu_2}$$

Thereafter is it decreased by a factor of μ_1 such that it is updated:

$$\Delta = \frac{\sigma_s}{\mu_1}$$

The theoretical progression of expected objective function values is predetermined by Δ for all temperature stages. If the observed average expected objective value at T_s is roughly equivalent to the theoretical value the algorithm is said to be in *equilibrium.* Alternatively if there is a significant gap between the two parameters should be tuned to minimize the gap.4 actions could be taken:

1. Increase the number of attempts for each temperature
2. Increase the current temperature
3. Choose a new smaller expected decrease in Δ
4. Terminate the algorithm & initialize a greedy algorithm

The author of the paper implemented a slight modification of the standard approach, given here:

Algorithm B.2: Simulated annealing with targeted average decrease in cost

```
1  dataset ← declareSystemData()
2  [current, currentObj ] ← generateRandomSolution(dataset)
3  [avgT_0, stdT_0] ← initialTemperature(current, currentObj, dataset)
4  T ← stdT_0
5  σ ← stdT_0
6  Δ ← σ/μ_2
7  negativeTemperature ← 0
8  reinitialising ← true
9  doGeometric ← false
10 while system not frozen do
11     Do the inner loop of SA and return the standard deviation     // Metropolis loop
12     σ ← standard deviation
13     if doGeometric = false then
14         if reinitialising = false then
15             if currentAverageCost/(previousAverageCost − Δ) > ζ then
16                 equilibriumNotReached ← equilibriumNotReached + 1
17             else
18                 equilibriumNotReached ← 0
19             end
20         end
21         if equilibriumNotReached > K_1 then
22             reinitialising ← true
23             equilibriumNotReached ← 0
24             T ← geometricCooling(λ_1, T)                    // reheating (λ_1 > 1)
25             Δ ← σ/μ_1
26         else if TΔ/σ^2 > 1 then
27             negativeTemperature ← negativeTemperature + 1
28             reinitialising ← true
29             if negativeTemperature < K_2 then
30                 T ← geometricCooling(λ_1, T)                // reheating (λ_1 > 1)
31                 Δ ← σ/μ_1
32             else
33                 doGeometric ← true
34             end
35         else
36             reinitialising ← false
37             previousAverageCost ← currentAverageCost
38             T ← TrikiCooling(Δ, σ, T)
39         end
40     else
41         T ← geometricCooling(λ_2, T)
42     end
43 end
```

```r
# ------------------------------------------------ Cooling Schemes ----------------------------------
Geo <- function(a, Ts) a*Ts

Huang <- function(Ts, lambda, sig_s) Ts * exp(-(lambda*Ts)/sig_s)

VanL <- function(Ts, delta, sig_s) Ts / (1 + ((log(1+delta))/(3*sig_s)) * Ts)

Triki <- function(Ts, Delta, sig_s) Ts * (1 - Ts*(Delta/sig_s))
```

Inner Metropolis Loop

The metropolis algorithm determines how many iterations are performed at each temperature. In our case, the inner loop terminates when any *one* of the *two* following conditions are satisfied:
1. A maximum of $12N$ solutions are accepted, or 2. A maximum of $100N$ solutions are attempted
Where N denotes the number of degrees of freedom in the problem. In our case: $N = n$.
Theoretical Lower Bounds

Theoretical lowerbounds are computed to help measure the performance achieved. Taken from the paper:

21-Unit System Computed with a weekly average reserve of $477.6MW$, disregarding the descrete unit capacity, maintenance window & crew constraints, a lower bound of the objective is then given by:

$$11'861'100\ MW^2$$

32-Unit IEEE-RTS inspired system Average weekly reserve level of $801MW$ yields a theoretical lower bound of:

$$33'363'252\ MW^2$$

Hyperparameters Selection

The thesis goes into a detailed examination of a wide range of possible hyper-parameters in order to determine an optimal configuration. The explanation is available in the thesis, the chosen values are given as:

21-Unit System

After great scrutinary & testing, the following hyper parameters were selected. Note that the initialization techniques are referred to as *AIM* (average increase method) & *SDM* (standard deviation method).

Geometric Cooling

$$\begin{aligned} \textit{initial temperature geometric cooling} &= AIM \\ \alpha &= 0.92 \\ T_{min} &= 1 \\ \textit{max attempts} &= 80n \\ \textit{neighbourhood size} &= 2n \\ \textit{iterations} &= 7000 \\ \textit{no. iterations without an improvement} &= 0.8I \end{aligned}$$

Huang Cooling

$$\begin{aligned} \textit{initial temperature Huang cooling} &= AIM \\ \lambda &= 0.6 \\ T_{min} &= 1 \\ \textit{max attempts} &= 100n \end{aligned}$$

Van Laarhoven Cooling

$$\begin{aligned} \textit{initial temperature Van Laarhoven cooling} &= AIM \\ \lambda &= 0.16 \\ T_{min} &= 1 \\ \textit{max attempts} &= 100n \end{aligned}$$

Triki Cooling

$$\begin{aligned} \textit{initial temperature Van Laarhoven cooling} &= SDM \\ \zeta &= 1.02 \\ \mu_2 &= 10 \\ \mu_1 &= 10 \\ T_{min} &= 1 \\ \textit{max attempts} &= 100n \\ \\ \textit{neighbourhood size} &= 2n \\ \textit{iterations} &= 7000 \\ \textit{no. iterations without an improvement} &= 0.8I \end{aligned}$$

32-Unit System Hyperparameters

All specification utilize:

$$T_{min} = 1$$

Geometric Cooling

$$\begin{aligned} \textit{initial temperature geometric cooling} &= AIM \\ \alpha &= 0.92 \\ T_{min} &= 1 \\ \textit{max attempts} &= 80n \end{aligned}$$

Huang Cooling

$$\begin{aligned} initial\ temperature\ Huang\ cooling &= AIM \\ \lambda &= 0.54 \\ T_{min} &= 1 \\ max\ attempts &= 100n \end{aligned}$$

Van Laarhoven Cooling

$$\begin{aligned} initial\ temperature\ Van\ Laarhoven\ cooling &= SDM \\ \delta &= 0.35 \\ T_{min} &= 1 \\ max\ attempts &= 90n \end{aligned}$$

Triki Cooling

$$\begin{aligned} initial\ temperature\ Van\ Laarhoven\ cooling &= SDM \\ \zeta &= 1.06 \\ \mu_2 &= 10 \\ \mu_1 &= 10 \\ T_{min} &= 1 \\ max\ attempts &= 90n \end{aligned}$$

Hyperparameters Summary

A summary of the hyperparameters chosen is given here:

	The 21-unit system		The 22-unit system		The IEEE-RTS inspired system	
Parameter	Classical	Ejection chain	Classical	Ejection chain	Classical	Ejection chain
Neighbourhood size	m	$2n$	m	$2n$	m	n
Iterations (I)	8 000	7 000	9 000	9 000	10 000	10 000
Number of iterations without improvement	$0.8I$	$0.8I$	$0.8I$	$0.8I$	$0.9I$	$0.8I$

Table 5.7: *Optimised parameter values for the random search heuristic.*

		The 21-unit system		The 22-unit system		The IEEE-RTS inspired system	
Cooling schedule	Parameter	Classical	Ejection chain	Classical	Ejection chain	Classical	Ejection chain
Geometric	Method for T_0	AIM	AIM	AIM	AIM	AIM	AIM
	α	0.95	0.92	0.97	0.96	0.95	0.92
	$max\_attempt$	$100n$	$100n$	$80n$	$80n$	$90n$	$80n$
Huang	Method for T_0	AIM	AIM	AIM	SDM	AIM	AIM
	λ	0.6	0.6	0.62	0.72	0.54	0.54
	$max\_attempt$	$100n$	$100n$	$100n$	$100n$	$100n$	$100n$
Van Laarhoven	Method for T_0	AIM	AIM	AIM	AIM	AIM	SDM
	δ	0.16	0.16	0.16	0.2	0.15	0.35
	$max\_attempt$	$90n$	$100n$	$80n$	$90n$	$90n$	$90n$
Triki	Method for T_0	SDM	SDM	SDM	SDM	SDM	SDM
	ζ	1.02	1.02	1.02	1.02	1.06	1.06
	μ_1	10	10	10	10	10	10
	μ_2	10	10	5	10	10	10
	$max\_attempt$	$100n$	$100n$	$100n$	$100n$	$100n$	$90n$

Table 5.8: *Optimised parameter values for the simulated annealing algorithm.*

Triki The Triki implementation requires some additional hyperparameters, that's selection is not detailed in the thesis. Examine algorithm B.2, which is used in implementing Triki in the thesis:

Algorithm B.2: Simulated annealing with targeted average decrease in cost

```
dataset ← declareSystemData()
[current, currentObj ] ← generateRandomSolution(dataset)
[avgT_0, stdT_0] ← initialTemperature(current, currentObj, dataset)
T ← stdT_0
σ ← stdT_0
Δ ← σ/μ_2
negativeTemperature ← 0
reinitialising ← true
doGeometric ← false
while system not frozen do
    Do the inner loop of SA and return the standard deviation      // Metropolis loop
    σ ← standard deviation
    if doGeometric = false then
        if reinitialising = false then
            if currentAverageCost/(previousAverageCost − Δ) > ζ then
                equilibriumNotReached ← equilibriumNotReached + 1
            else
                equilibriumNotReached ← 0
            end
        end
        if equilibriumNotReached > K_1 then
            reinitialising ← true
            equilibriumNotReached ← 0
            T ← geometricCooling(λ_1, T)                          // reheating (λ_1 > 1)
            Δ ← σ/μ_1
        else if TΔ/σ^2 > 1 then
            negativeTemperature ← negativeTemperature + 1
            reinitialising ← true
            if negativeTemperature < K_2 then
                T ← geometricCooling(λ_1, T)                      // reheating (λ_1 > 1)
                Δ ← σ/μ_1
            else
                doGeometric ← true
            end
        else
            reinitialising ← false
            previousAverageCost ← currentAverageCost
            T ← TrikiCooling(Δ, σ, T)
        end
    else
        T ← geometricCooling(λ_2, T)
    end
end
```

The implementation requires one to specify a number of additional hyperparameters:

$$K_1 \in [1,4] : number\ of\ metropolis\ chains$$
$$K_2 \in [2,10] : unit\ runs$$
$$\lambda_1 \in [1.4,4] : increase\ temperature$$

The paper complete ignores there parameters. Note that λ_1 is used in implementing *geometric cooling* but is actually used for *geometric heating* as it is larger than 1.

These parameters are selected by those suggested in the original paper - reference [79] in the thesis - given by:

$$K_1 = 2$$
$$K_2 = 4$$
$$\lambda_1 = 2$$

Implementation

Now that we have detailed the implementation's workings and defined the data as well as some of the core functions, we can implement the full solution. The results achieved do not mimic those found in the paper, though implementation is identical.

Given the stochastic nature of the algorithm, one ought to run the same experiment many times in order to average the results achieved in assessing the algorithms performance. The paper repeats 20 experiments over 50 randomly generated starting configurations, we cannot replicate this frequency due to computational limits, however here we setup the implementation to allow for this if someone wishes to compute all 50 implementations 20 times.

Random Initialization Configurations

Here we load the data & generate the 50 starting configurations, for both the 21 & 32 unit cases.

```
# ------------------------------------------------------ 21-Unit Case ---------------------------------

# ----- Data -----x
load_data(IEEE_RTS_system = FALSE)
el <- cbind(e,l)

# ----- 50 Random Inits -----x
set.seed(10)
inits_21 <- lapply(1:50, init_solution, size=size, m=m, e=e, l=l)



# ------------------------------------------------------ 32-Unit Case ---------------------------------

# ----- Data -----x
load_data(IEEE_RTS_system = TRUE)
el <- cbind(e,l)

# ----- 50 Random Inits -----x
```

```r
set.seed(10)
inits_32 <- lapply(1:50, init_solution, size=size, m=m, e=e, l=l)
```

Unfortunately, due to computational limits, we cannot implement all 50 attempts, & will simple implement the first 2.

These same starting configurations will be used in the simulated annealing application - as done in the paper. Now we are ready to implement the random search over the various specifications.

Random Search (Baseline)

We begin by implementing the random search algorithm described here:

Algorithm B.1: The GMS random search heuristic with classical neighbourhood

Input: A power system scenario for which to solve the generator maintenance scheduling problem

Output: The best maintenance schedule found

```
1  dataset ← declareSystemData()
2  [current, currentObj] ← generateRandomSolution(dataset)
3  [incumbent, incumbentObj] ← [current, currentObj]
4  iterationCounter ← 0
5  nonImproveCounter ← 0
6  while (iterationCounter < maxIteration) and (nonImproveCounter < maxNonImprove)
   do
7      bestNeighbour ← ∅
8      bestNeighbourObj ← some very large number
9      moves ← createClassicalNeighbourhoodList (n, e, ℓ, W_ext)
10     moves ← randShuffle(moves)
11     for neighbourCounter ← 1 to neighbourhoodSize do
12         neighbour ← current
13         Apply moves(neighbourCounter) on neighbour to create new neighbour
14         P ← checkFeasibilityAndCalculatePenalty(neighbour, dataset)
15         Calculate neighbourObj
16         neighbourObj ← neighbourObj + P
17         if neighbourObj < bestNeighbourObj then
18             [bestNeighbour, bestNeighbourObj] ← [neighbour, neighbourObj]
19         end
20     end
21     if bestNeighbourObj < incumbentObj then
22         [incumbent, incumbentObj] ← [bestNeighbour, bestNeighbourObj]
23         nonImproveCounter ← 0
24     else
25         nonImproveCounter ← nonImproveCounter + 1
26     end
27     [current, currentObj] ← [bestNeighbour, bestNeighbourObj]
28     iterationCounter ← iterationCounter + 1
29 end
```

Algorithm B.1: The GMS random search heuristic with classical neighbourhood

Input: A power system scenario for which to solve the generator maintenance scheduling problem

Output: The best maintenance schedule found

```
1  dataset ← declareSystemData()
2  [current, currentObj] ← generateRandomSolution(dataset)
3  [incumbent, incumbentObj] ← [current, currentObj]
4  iterationCounter ← 0
5  nonImproveCounter ← 0
6  while (iterationCounter < maxIteration) and (nonImproveCounter < maxNonImprove)
   do
7      bestNeighbour ← ∅
8      bestNeighbourObj ← some very large number
9      moves ← createClassicalNeighbourhoodList (n, e, ℓ, W_ext)
10     moves ← randShuffle(moves)
11     for neighbourCounter ← 1 to neighbourhoodSize do
12         neighbour ← current
13         Apply moves(neighbourCounter) on neighbour to create new neighbour
14         P ← checkFeasibilityAndCalculatePenalty(neighbour, dataset)
15         Calculate neighbourObj
16         neighbourObj ← neighbourObj + P
17         if neighbourObj < bestNeighbourObj then
18             [bestNeighbour, bestNeighbourObj] ← [neighbour, neighbourObj]
19         end
20     end
21     if bestNeighbourObj < incumbentObj then
22         [incumbent, incumbentObj] ← [bestNeighbour, bestNeighbourObj]
23         nonImproveCounter ← 0
24     else
25         nonImproveCounter ← nonImproveCounter + 1
26     end
27     [current, currentObj] ← [bestNeighbour, bestNeighbourObj]
28     iterationCounter ← iterationCounter + 1
29 end
```

A part from simply accepting all moves with a probability of 1. one important detail, that turns out to be of utmost importance, differentiates this implementation from the simulated annealing implementation.

These random search implementations search the entire neighbourhood of the current solution, iterating over n or nm neighbours for each move. This adds significant computational cost, however in our implementations appears even more useful than the 'intelligent' simulated annealing cooling structure.

This approach, more rigorously explores the solution space, increases the likelihood of optimizing on a local sub-search within the main search The fact that every proposed move is accepted will inevitably produce a random walk, however by executing this *deep search* within each iteration finds good local optima & produces great results. This is omitted from the main SA implementation - presumably to prevent premature convergence - though some adaptation (possibly a smaller) neighbourhood size, could be favourable. In fact, this is what we see with the hybrid solution, in which local search is performed over incumbent solution without updating the current solution.

A standard baseline is to compare the results achieved with random search. Here we implement random search, for the same 50 random starting conditions, over both the *classical* & *ejection chain* operators. We, further, need to determine the number of iterations. We implement the same number of iterations as that which was selected for each variation, thus see the tables above for the number of iterations per configuration.

As such we implement $4 = 2 \times 2$ random searches: - *21-Unit* utilizing the *Classical* Local Search Operator - *21-Unit* utilizing the *Ejection Chain* Local Search Operator - *32-Unit* utilizing the *Classical* Local Search Operator - *32-Unit* utilizing the *Ejection Chain* Local Search Operator

21-Unit Classical Operator Random Search

The algorithm below is fully parameterizable in that the user can specify:

- max iterations
- max iterations without improvement
- neighbourhood size
- move operator
- random starting solutions (computed above)

Thus, the same function can be used to generate all of the random baseline solutions. Because of computational limits, only a single run will be used to compute the random baselines. Further, knitting this document is an issue & often causes R to crash. As such the computation is not performed at *knit* time, but instead we compute the algorithm, save the result & read in the results to display on knit time.

This practice of saving & reloading results is repeated as many configurations take many hours to compute & as such computing on *knit time* is impractical. The local neighbourhood search is an available parameter: the use can specify the size α of the neighbourhood, the algorithm will then compute α neighbours for each temperature value & select the best alternative.

This feature simply provides a more thorough search, as such (& again due to computational limits) we limit the neighbourhood size to $\alpha = 5$ for the random search implementations as these are simply baseline runs.

The more appropriate α for this specification (21-unit classical operator) would be $\alpha = m$ as seen in the table above.

Max iterations is taken to be the same as the best implementation, in this case that is 8000. Similarly we set *max iterations with no improvement* to 8000.

The below implementation can be used for all random searches, we attempt to keep algorithms universal as often as possible.

Important: Note that the algorithm takes a list of initial solutions as input, not a single random initial solution. As such it also returns a list of results (the first of which, for example, is accessible by results[[1]]). This is done so that one could, in theory, pass all 50 starting configurations and receive all results in one clean run. We cannot do this due to computational requirements, & pass a single run.

```
# ---- hyper-parameter config ----


random_search <- function(max_iters, max_non_improve, neighbourhood_size, move_operator, random_inits, r
  # ------ prints ------x
  print(paste('----------------- Running Computation:', name, '-----------------'))
```

```r
main_start <- Sys.time()

# ------ final results ------x
final <- list()

results <- c()
for (inits in (1:length(random_inits))) {

  # ------ prints ------x
  print(paste('trail: ', inits, '/', length(random_inits), sep=''))

  # ------ Inits: for each random init ------x
  start <- Sys.time()

  non_improve_count <- 0

  # ------ init solution ------x
  x <<- random_inits[[inits]]
  y <<- compute_y_binary_matrix()
  res <- compute_penalty()
  incumbent_sol <- current_sol <- x
  incumbent_score <- current_score <- res$Q_total

  # ------ store results ------x
  solutions <- c()
  results <- c()
  scores <- c()
  Pens <- c()
  Pws <- c()
  Pls <- c()
  Pcs <- c()
  Pes <- c()
  Qs <- c()
  Q_totals <- c()
  feasibility <- c()

  iter <- 0
  while ((iter < max_iters) & (non_improve_count < max_non_improve)) {
    # ----- progress report -----x
    if ((iter %% (8000/5)) == 0) print(paste((iter %/% (8000/5)), '/5 completed', sep = ''))



    best_neighbour <- 0
    best_neighbour_Q <- 1e+100

    # ------ generate neighbours ------x
    neighbours <- lapply(1:neighbourhood_size, move_operator, sol=x, el=el)


    # ------ evaluate neighbours ------x
    neighbour_res <- c()
    evaluate_x_y <- function(index, neighbours=neighbours) {
```

```
      x <<- neighbours[[index]]
      y <<- compute_y_binary_matrix()
      return(compute_penalty())
    }
    neighbour_res <- lapply(1:neighbourhood_size, evaluate_x_y, neighbours=neighbours)


    # ------ find best neighbour ------x
    Q_vals <- c()
    for (i in neighbour_res) Q_vals <- c(Q_vals, i$Q_total)
    best_ind <- which(Q_vals == min(Q_vals))[1]


    # ------ update incumbent ------x
    if (Q_vals[[best_ind]] < incumbent_score) {
      incumbent_sol <- neighbours[[best_ind]]
      incumbent_score <- Q_vals[[best_ind]]
    } else {
      non_improve_count <- non_improve_count + 1
    }

    # ------ update current ------x
    current_sol <- neighbours[[best_ind]]
    current_score <- Q_vals[[best_ind]]
    res <- neighbour_res[[best_ind]]


    # ------ store results ------x
    iter <- iter + 1
    solutions[[iter]] <- current_sol
    scores[[iter]] <- current_score
    # results[[iter]] <- res
    Pens[[iter]] <- res$P
    Pws[[iter]] <- res$Pw
    Pls[[iter]] <- res$Pl
    Pcs[[iter]] <- res$Pc
    Pes[[iter]] <- res$Pe
    Qs[[iter]] <- res$Q
    Q_totals[[iter]] <- res$Q_total
    feasibility[[iter]] <- res$feasible

    }
    end <- Sys.time()
    runtime <- end - start

    # ------ update final ------x
    fin <- list(inits=inits, name=name, runtime=runtime, incumbent_sol=incumbent_sol, incumbent_score=
                non_improve_count=non_improve_count, iter=iter, solutions=solutions, scores=scores,
                Pens=Pens, Pws=Pws, Pls=Pls, Pcs=Pcs, Pes=Pes, Qs=Qs, Q_totals=Q_totals, feasibility=
    final[[inits]] <- fin
}

# ------ prints ------x
```

```
  main_time <- Sys.time() - main_start
  print('5/5 completed')
  print(paste('--------------------------------------------------------------------'))
  print(main_time)

  print(paste('----------------- Complete Computation:', name, '-----------------'))

  # ------ RETURN results ------x
  return(final)
}


# ----- Data -----x
load_data(IEEE_RTS_system = FALSE)
el <- cbind(e,l)

# ----- Random Run -----x
# results_21_classical <- random_search(max_iters=8000, max_non_improve=1*8000, random_inits=inits_21[1
#                                       neighbourhood_size=5, move_operator=classical_operator, name='2


# ---- save results ----x
# saveRDS(results_21_classical, file = "./cached computed results/results_21_classical.rds")

# load results
results_21_classical <- readRDS(file = "./cached computed results/results_21_classical.rds")
```

In order to save on computation when knitting the final document, may results are cached away & reloaded. Note that the code to compute the run is commented out, as it has been run, saved, & reloaded.

This is done for all the random runs as it is just simply not feasible to compute all runs on knit time as the computer often crashes.

Understanding the results data-structure: The output datastructure is fairly complex, as such here's a brief description of how it works. The function returns a list of lists, one for each random start used. Thus in the full instance there will be 50 sublists however in our instance we have only one.

Each sub-list contains the details relating to that specific run. The data available from each run is show below.Most of which are self-explanatory, the most important are:

- *incumbent_sol*: which gives the final incumbent solution
- *incumbent_score*: gives the final incumbent score
- *Q_totals*: gives a list of objective values of the current solutions

Both the aggregated & individual weights are returned, though importantly one should note that the penalties are return AFTER the weights have been applied: one should consider this if analysis the model sensitivity to weights.

```
random_run_summary <- function(run) {
  print(paste('Run name:', run[[1]]$name))

  print(paste('Number of random inits used: ',length(run)))
  print('_____________________________________________')
```

```
  print('available data:')
  print(names(run[[1]]))
  print('_____________________________________________')
  print('Incumbent solution: ')
  print(run[[1]]$incumbent_sol)

  print('_____________________________________________')
  print(paste('Incumbent score: ', round(run[[1]]$incumbent_score), sep=''))

}


random_run_summary(results_21_classical)
```

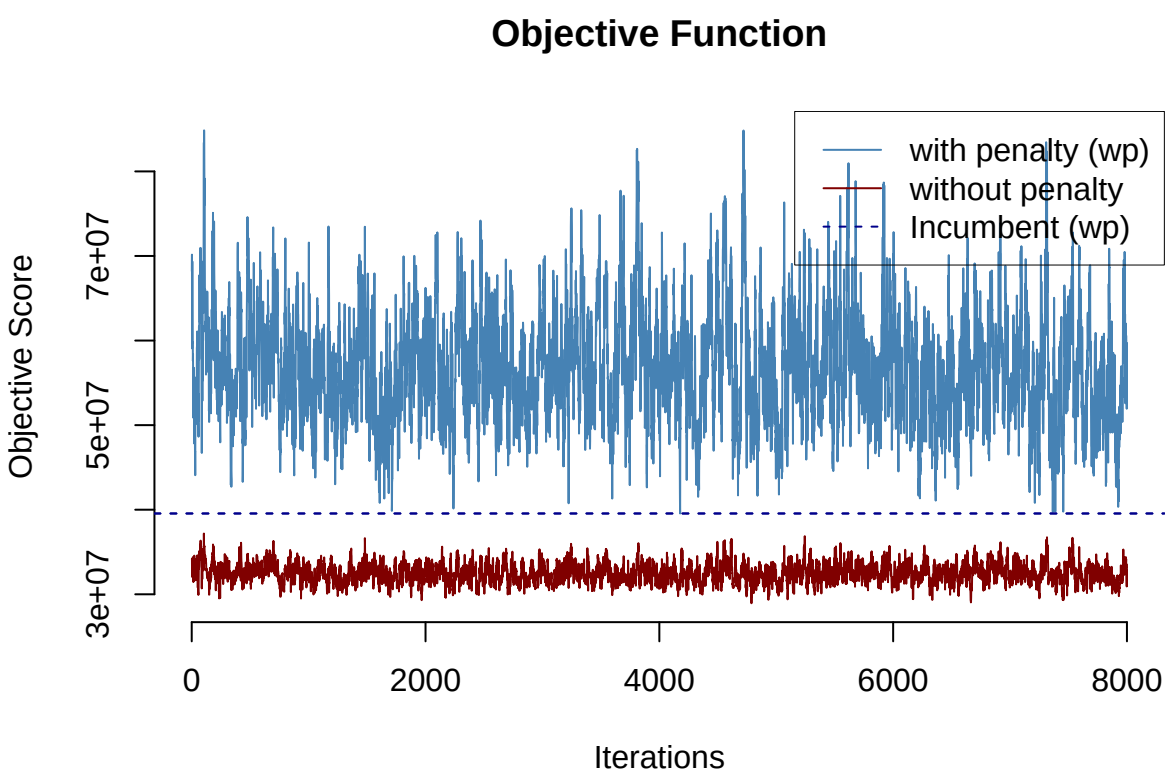
```
## [1] "Run name: 21-unit Classical Operator Search"
## [1] "Number of random inits used:  1"
## [1] "_____________________________________________"
## [1] "available data:"
##  [1] "inits"             "name"              "runtime"
##  [4] "incumbent_sol"     "incumbent_score"   "non_improve_count"
##  [7] "iter"              "solutions"         "scores"
## [10] "Pens"              "Pws"               "Pls"
## [13] "Pcs"               "Pes"               "Qs"
## [16] "Q_totals"          "feasibility"
## [1] "_____________________________________________"
## [1] "Incumbent solution: "
##  [1]  2 47 25 19 28  2 15 39 10 14 24 45  6  8 21 12 33 35 43 37 10
## [1] "_____________________________________________"
## [1] "Incumbent score: 39554195"
```

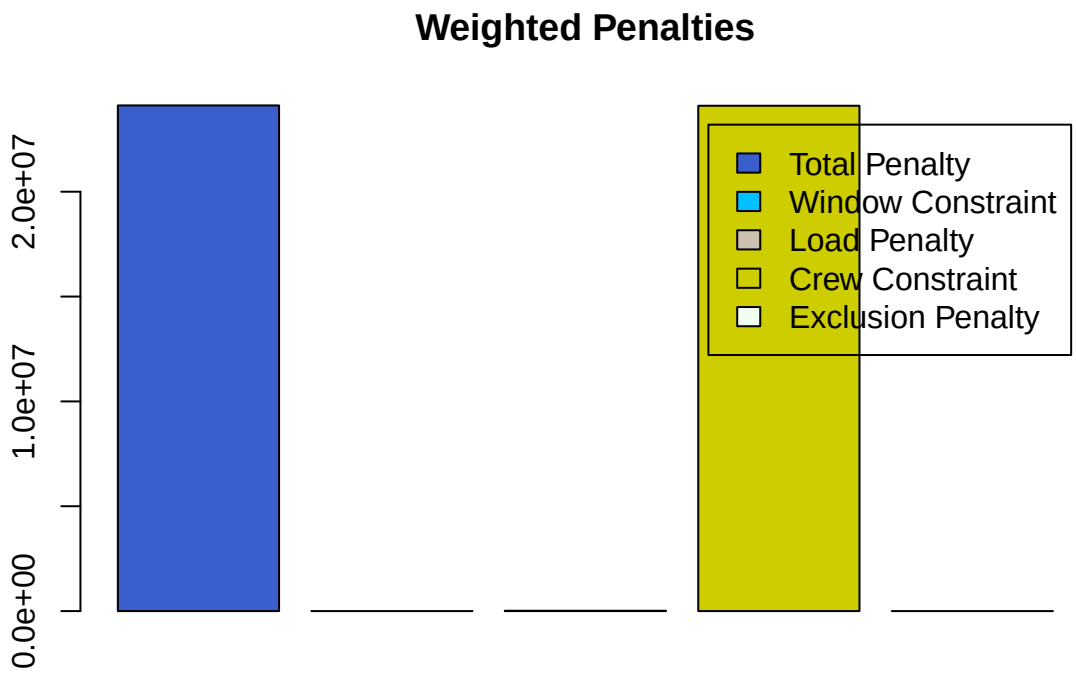
It would also be nice to have a function that plots the various metrics of interest, that we can apply to each random run. Here we define such a function:

```
compute_summary <- function(results_1_run) {
  res <- results_1_run

  print('- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -')
  print(paste('Best Objective: '))
  print(res$incumbent_sol)
  print('- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -')
  print(paste('Best Objective: ', res$incumbent_score))
  print('- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -')
  print(paste('Proportion of Infeasible Solutions: ', mean(unlist(res$feasibility))))
  print('- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -')
  print(res$runtime)
  print('- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -')


  # ----- Graph 1: Solution Results -----x
  plot(1:length(res$Q_totals), res$Q_totals, frame=F, 'l', col='steelblue', main='Objective Function', y
       xlab='Iterations', ylim = c(min(unlist(res$Qs))-100, max(unlist(res$Q_totals))+100))
  lines(1:length(res$Q_totals), res$Qs,  'l', col='#800000')
  abline(h = min(unlist(res$Q_totals)), lty=2, col='darkblue')
```

```
  abline(h = res$incumbent_score, lty=2, col='darkblue')
  legend('topright', col = c('steelblue', '#800000', 'darkblue'),
         legend = c('with penalty (wp)', 'without penalty', 'Incumbent (wp)'), lty = c(1,1,2), box.lwd =



  # ----- Graph 2: Average Penalty -----x
  vals <- c(mean(unlist(res$Pens)), mean(unlist(res$Pws)), mean(unlist(res$Pls)), mean(unlist(res$Pcs))
  vals[is.na(vals)] <- 0
  names <- c('Total Penalty', 'Window Constraint', 'Load Penalty', 'Crew Constraint', 'Exclusion Penalty
  barplot(height = vals, col = sample(colors(), 5), legend.text = names, main='Weighted Penalties')

}

compute_summary(results_21_classical[[1]])
```

```
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Best Objective: "
##  [1]  2 47 25 19 28  2 15 39 10 14 24 45  6  8 21 12 33 35 43 37 10
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Best Objective:  39554195.0375"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Proportion of Infeasible Solutions:  0"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## Time difference of 6.858224 mins
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
```


Objective Function
with penalty (wp)
without penalty
Incumbent (wp)
Objective Score
3e+07
5e+07
7e+07
0
2000
4000
6000
8000
Iterations



We expect the *window constraint* to be 0 as it implementation only proposes feasible solutions (w.r.t this constrant). We also expect the *exclusion constraint* to be 0 due to the fact that the 21 *case* implementation does not have exclusion requirements. We do not, however, expect the load penalty to be 0, it is not zero, it is simple far smaller than the others. This is bring into question the weights chosen by the paper.

Other Configurations

Similarly we compute a single random run over the 3 other specifications:

- 21-Unit Ejection Chain Operator Random Search
- 32-Unit Classical Operator Random Search
- 32-Unit Ejection Chain Operator Random Search

which will then be used as baseline of comparison agree other runs. Additionally, as this random walk is essentially a stationary process, one needs not concern themselves with running for excessively long, as such all random search variants will run for 8000 iterations & no more. It simply serves as a baseline.

Due to computational requirements, we will only run each variation once & the neighbourhood size is limited to 5.

21-Unit Ejection Chain Operator Random Search

```
# ----- Data -----x
load_data(IEEE_RTS_system = FALSE)
```

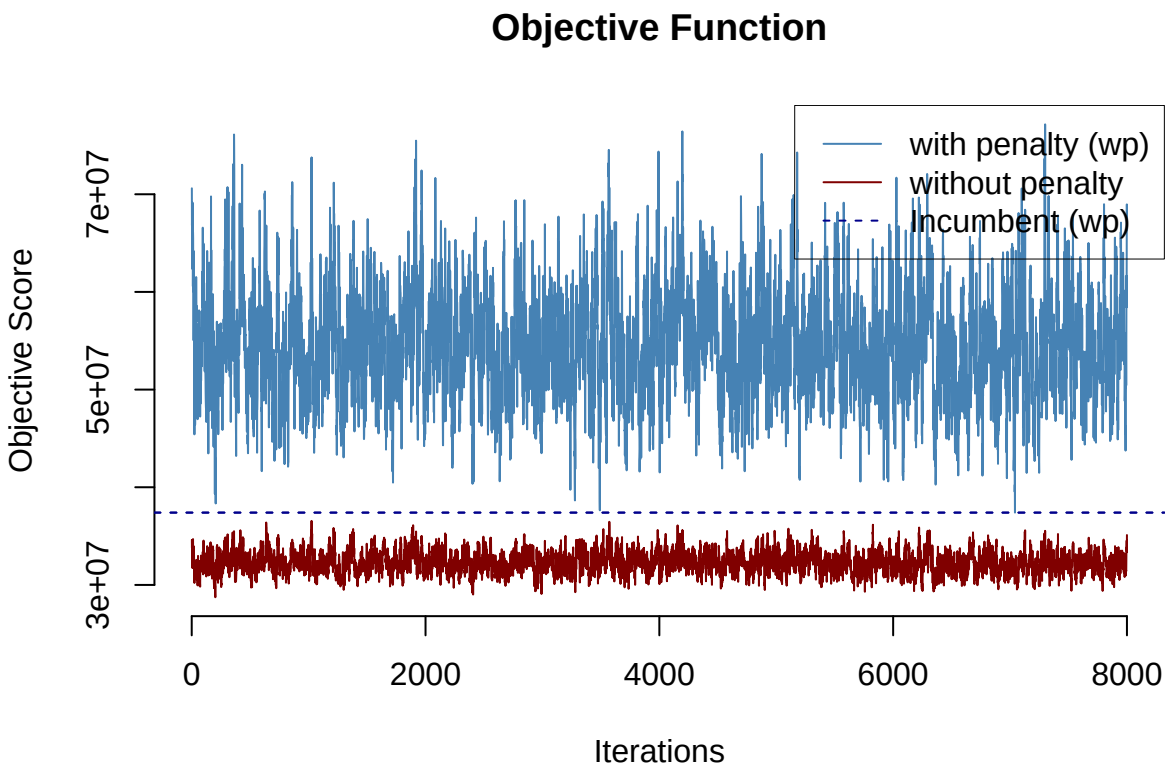
```r
el <- cbind(e,l)

# ----- Random Run -----x
# results_21_eject <- random_search(max_iters=8000, max_non_improve=1*8000, random_inits=inits_21[1],
#                                   neighbourhood_size=5, move_operator=ejection_chain_operator,
#                                   name='21-unit Ejection Chain Operator Search')



# ---- save results ----x
# saveRDS(results_21_eject, file = "./cached computed results/results_21_eject.rds")

# ---- load results ----x
results_21_eject <- readRDS(file = "./cached computed results/results_21_eject.rds")
```

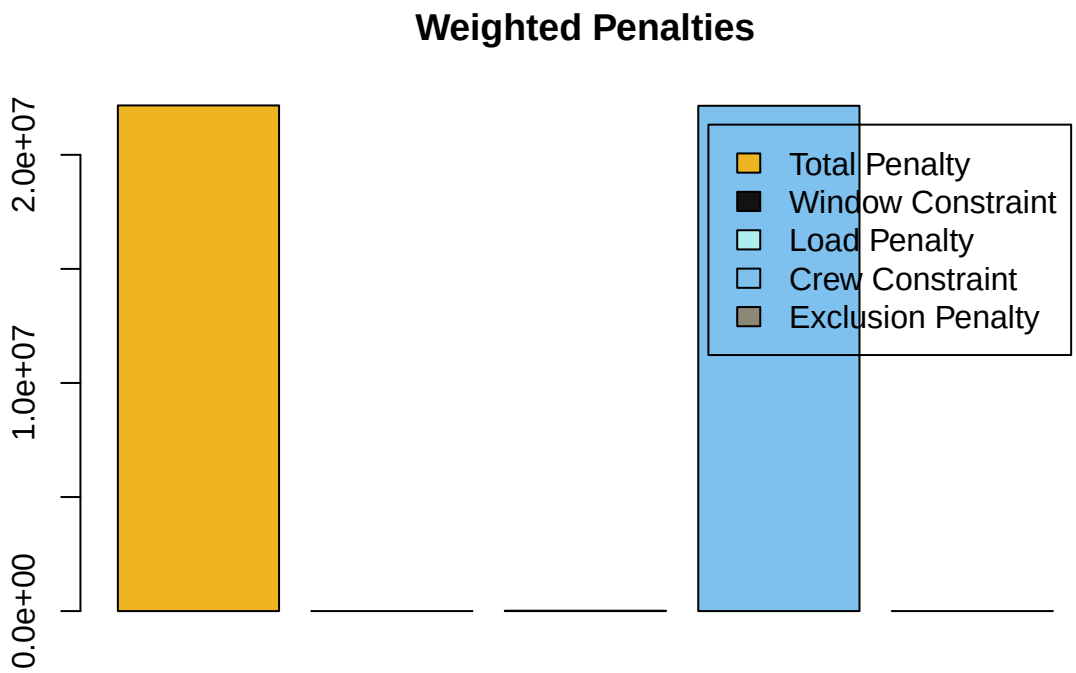
```r
random_run_summary(results_21_eject)
```

```
## [1] "Run name: 21-unit Ejection Chain Operator Search"
## [1] "Number of random inits used:  1"
## [1] "_____________________________________________"
## [1] "available data:"
##  [1] "inits"             "name"              "runtime"
##  [4] "incumbent_sol"     "incumbent_score"   "non_improve_count"
##  [7] "iter"              "solutions"         "scores"
## [10] "Pens"              "Pws"               "Pls"
## [13] "Pcs"               "Pes"               "Qs"
## [16] "Q_totals"          "feasibility"
## [1] "_____________________________________________"
## [1] "Incumbent solution: "
##  [1]  8 42 15 25 27  2 22 36  3 22 26 50 13  4 20 17 28 32 48 44 10
## [1] "_____________________________________________"
## [1] "Incumbent score: 37402896"
```

```r
compute_summary(results_21_eject[[1]])
```

```
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Best Objective: "
##  [1]  8 42 15 25 27  2 22 36  3 22 26 50 13  4 20 17 28 32 48 44 10
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Best Objective:  37402896.465"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Proportion of Infeasible Solutions:  0"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## Time difference of 6.832856 mins
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
```


Objective Function
with penalty (wp)
without penalty
Incumbent (wp)
Objective Score
7e+07
5e+07
3e+07
0
2000
4000
6000
8000
Iterations



32-Unit Classical Operator Random Search

```
# ----- Data -----x
load_data(IEEE_RTS_system = TRUE)
el <- cbind(e,l)

# ----- Random Run -----x
# results_32_classical <- random_search(max_iters=8000, max_non_improve=1*8000, random_inits=inits_32[1
#                                       neighbourhood_size=5, move_operator=ejection_chain_operator,
#                                       name='32-unit Classical Operator Search')


# ---- save results ----x
# saveRDS(results_32_classical, file = "./cached computed results/results_32_classical.rds")

# ---- load results ----x
results_32_classical <- readRDS(file = "./cached computed results/results_32_classical.rds")
```

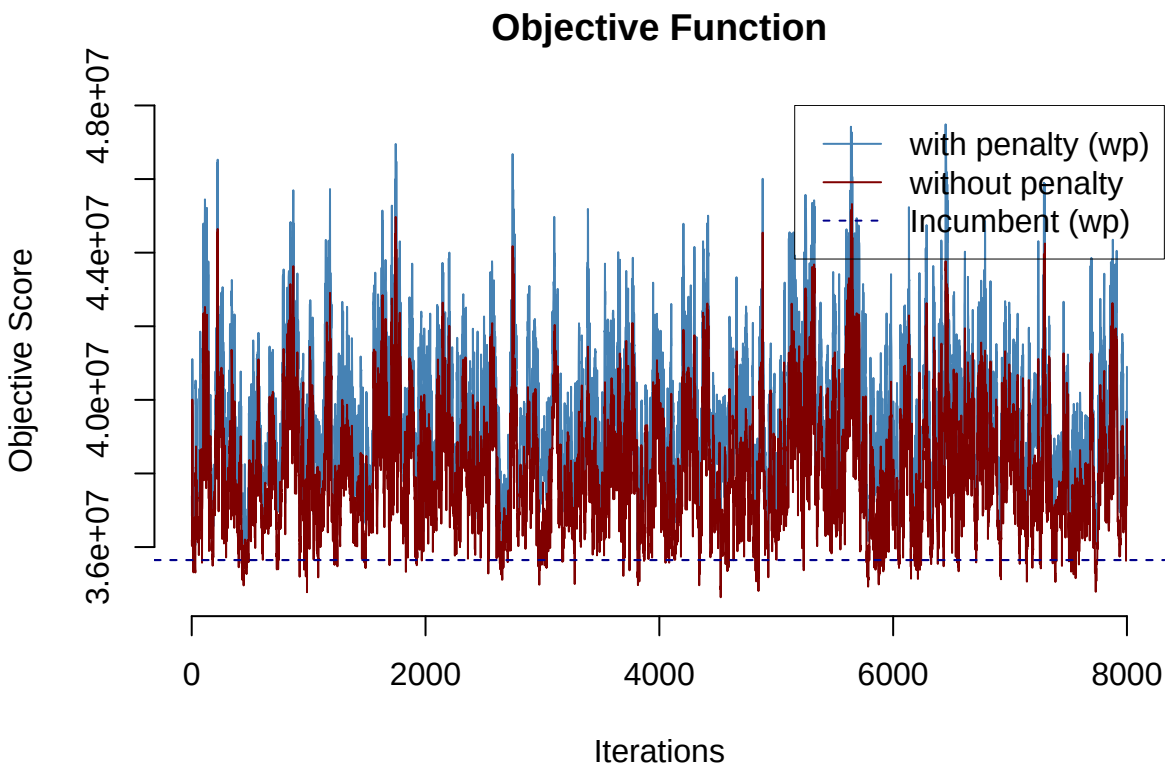
```
random_run_summary(results_32_classical)
```

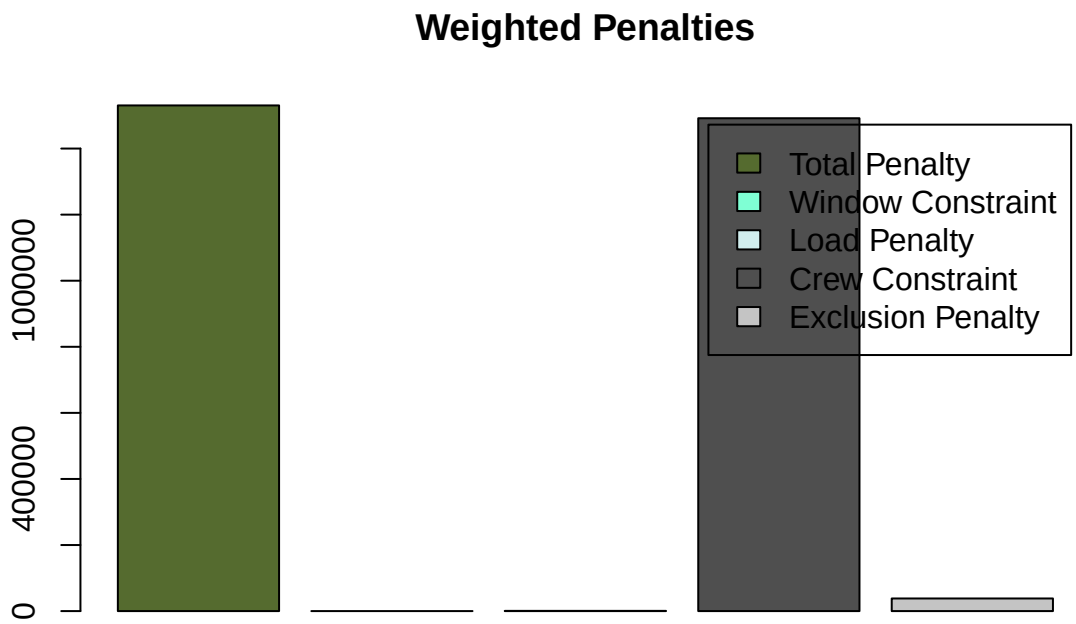
```
## [1] "Run name: 32-unit Classical Operator Search"
## [1] "Number of random inits used:  1"
## [1] "_____________________________________________"
```

```
## [1] "available data:"
##  [1] "inits"             "name"              "runtime"
##  [4] "incumbent_sol"     "incumbent_score"   "non_improve_count"
##  [7] "iter"              "solutions"         "scores"
## [10] "Pens"              "Pws"               "Pls"
## [13] "Pcs"               "Pes"               "Qs"
## [16] "Q_totals"          "feasibility"
## [1] "_____________________________________________"
## [1] "Incumbent solution: "
##  [1] 17 20  4 38 19 39 24 50 25 14 41  2 21 46 48 36 36 42 45 16 39 10 29  8  8
## [26]  7 51 11 11 20 26 34
## [1] "_____________________________________________"
## [1] "Incumbent score: 35646186"
```

```
compute_summary(results_32_classical[[1]])
```

```
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Best Objective: "
##  [1] 17 20  4 38 19 39 24 50 25 14 41  2 21 46 48 36 36 42 45 16 39 10 29  8  8
## [26]  7 51 11 11 20 26 34
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Best Objective:  35646186"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Proportion of Infeasible Solutions:  0"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## Time difference of 7.113293 mins
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
```


Objective Function
Objective Score
3.6e+07
4.0e+07
4.4e+07
4.8e+07
with penalty (wp)
without penalty
Incumbent (wp)
0
2000
4000
6000
8000
Iterations



We do expect an exclusion constraint violation in the 32 unit case, this is confirmed in our findings.

32-Unit Ejection Chain Operator Random Search

```
# ----- Data -----x
load_data(IEEE_RTS_system = TRUE)
el <- cbind(e,l)

# ----- Random Run -----x
# results_32_eject <- random_search(max_iters=8000, max_non_improve=1*8000, random_inits=inits_32[1],
#                                   neighbourhood_size=5, move_operator=ejection_chain_operator,
#                                   name='32-unit Ejection Chain Operator Search')



# ---- save results ----x
# saveRDS(results_32_eject, file = "./cached computed results/results_32_eject.rds")

# ---- load results ----x
results_32_eject <- readRDS(file = "./cached computed results/results_32_eject.rds")
```

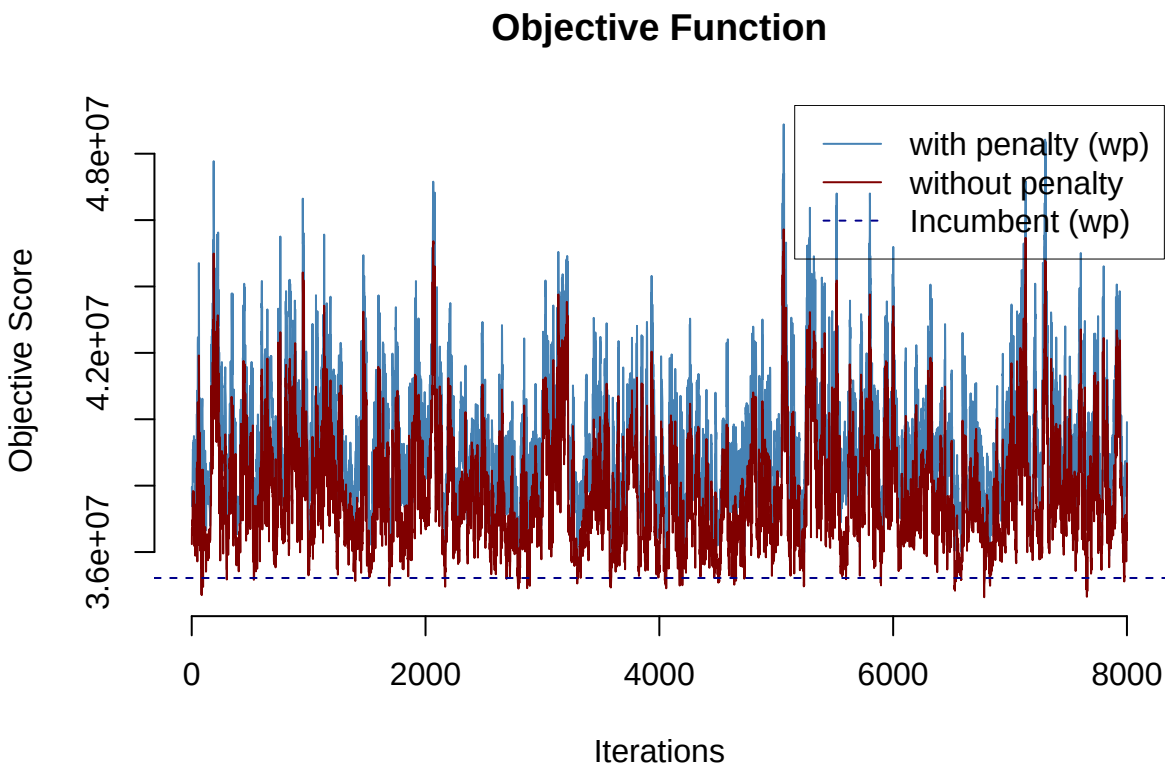
```
random_run_summary(results_32_eject)
```

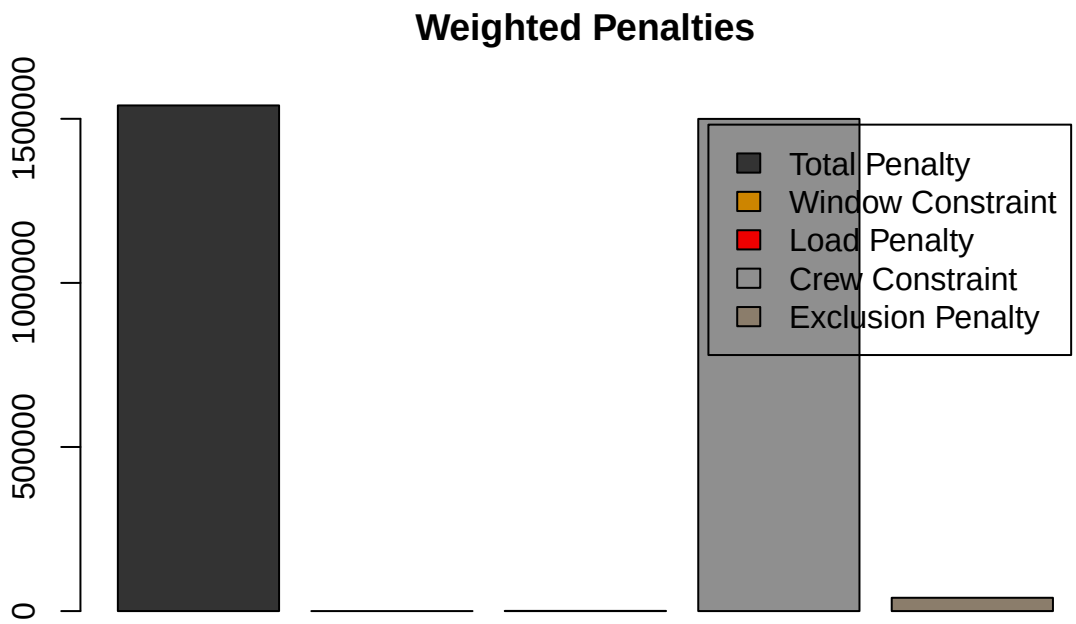
```
## [1] "Run name: 32-unit Ejection Chain Operator Search"
```

```
## [1] "Number of random inits used:  1"
## [1] "_____________________________________________"
## [1] "available data:"
##  [1] "inits"             "name"              "runtime"
##  [4] "incumbent_sol"     "incumbent_score"   "non_improve_count"
##  [7] "iter"              "solutions"         "scores"
## [10] "Pens"              "Pws"               "Pls"
## [13] "Pcs"               "Pes"               "Qs"
## [16] "Q_totals"          "feasibility"
## [1] "_____________________________________________"
## [1] "Incumbent solution: "
##  [1]  1  5 13 42 10 34 11 31 49 26  1 21  2 40 32  6 48 16 41 14 38 10 31 44 25
## [26] 46 18 29 24 19  7 36
## [1] "_____________________________________________"
## [1] "Incumbent score: 35218314"
```

```
compute_summary(results_32_eject[[1]])
```

```
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Best Objective: "
##  [1]  1  5 13 42 10 34 11 31 49 26  1 21  2 40 32  6 48 16 41 14 38 10 31 44 25
## [26] 46 18 29 24 19  7 36
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Best Objective:  35218314"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Proportion of Infeasible Solutions:  0"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## Time difference of 7.032786 mins
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
```


Objective Function
Objective Score
4.8e+07
4.2e+07
3.6e+07
with penalty (wp)
without penalty
Incumbent (wp)
0
2000
4000
6000
8000
Iterations



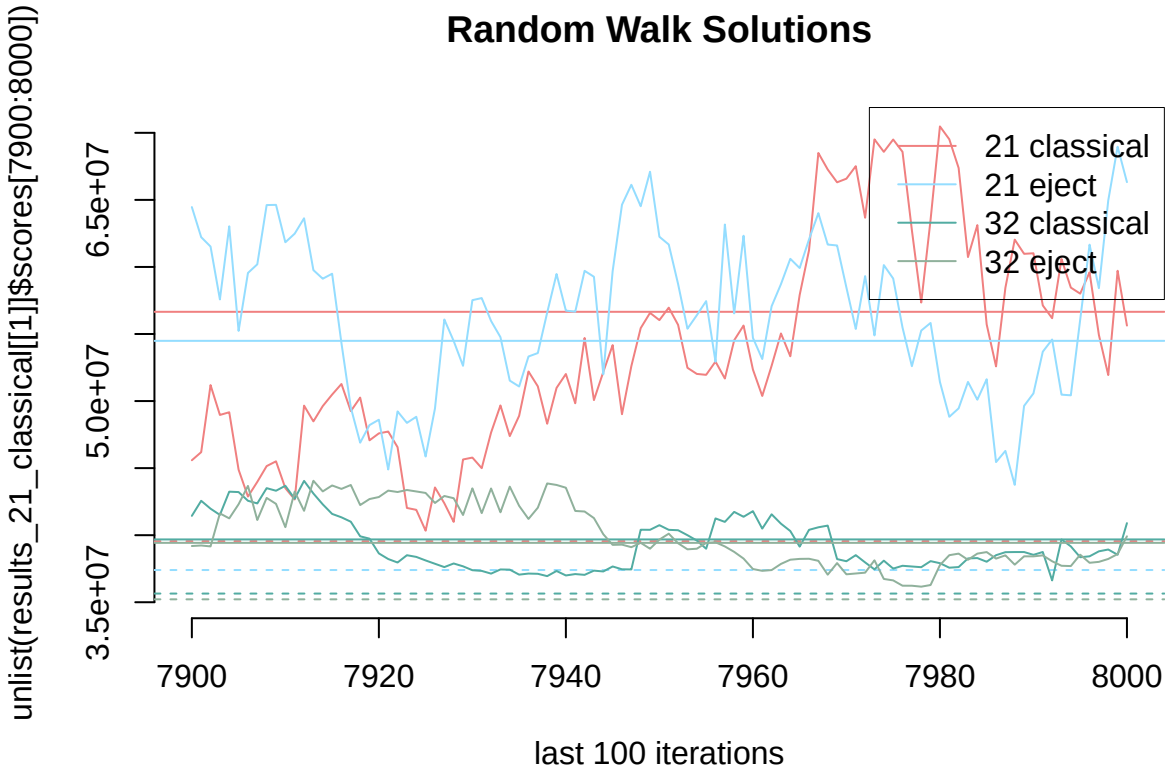
Compare the 4 Random Searches

The last 100 iterations of each random walk is graphed, as well as:

- the dashed lines denote the incumbent solution of each run
- the solid lines denote the average solution of each run

```
plot(7900:8000, unlist(results_21_classical[[1]]$scores[7900:8000]), 'l', col='#f08080', frame=F,
     main='Random Walk Solutions', xlab = 'last 100 iterations',
     ylim = c(
       min(results_21_classical[[1]]$incumbent_score, results_21_eject[[1]]$incumbent_score,
           results_32_classical[[1]]$incumbent_score, results_32_eject[[1]]$incumbent_score),
       max(unlist(results_21_classical[[1]]$scores[7900:8000]), unlist(results_21_eject[[1]]$scores[790
           unlist(results_32_classical[[1]]$scores[7900:8000]), unlist(results_32_eject[[1]]$scores[790
    ))
abline(h=results_21_classical[[1]]$incumbent_score, col='#f08080', lty=2)
abline(h=mean(unlist(results_21_classical[[1]]$scores), na.rm = T), col='#f08080', lty=1)

lines(7900:8000, results_21_eject[[1]]$scores[7900:8000], col='#94ddff')
abline(h=results_21_eject[[1]]$incumbent_score, col='#94ddff', lty=2)
abline(h=mean(unlist(results_21_eject[[1]]$scores), na.rm = T), col='#94ddff', lty=1)

lines(7900:8000, results_32_classical[[1]]$scores[7900:8000], col='#52aca2')
abline(h=results_32_classical[[1]]$incumbent_score, col='#52aca2', lty=2)
abline(h=mean(unlist(results_32_classical[[1]]$scores), na.rm = T), col='#52aca2', lty=1)
```

```
lines(7900:8000, results_32_eject[[1]]$scores[7900:8000], col='#90b19c')
abline(h=results_32_eject[[1]]$incumbent_score, col='#90b19c', lty=2)
abline(h=mean(unlist(results_32_eject[[1]]$scores), na.rm = T), col='#90b19c', lty=1)


legend('topright', legend = c('21 classical', '21 eject', '32 classical', '32 eject'),
       col = c('#f08080', '#94ddff', '#52aca2', '#90b19c'), lty=1, box.lwd = 0)
```



The ejection chain method, which incorporates information about the entire solution structure, appears to yield similar results. Although the searches are random walks, decent optima are found.

Best Paper Implementation

As per the paper, the best known solution was given by utilizing:

1. The *Van Laarhoven* cooling schedule
2. *Ejection Chain Search Operator*
3. *Hybrid Solution*

As such, here we implement this exact specification for both 21 & 32 unit cases. The function to implement such as algorithm is adapted from the that which is used above, though hybridization is added. Local search is performed whenever a new incumbent solution is updated.

The function is written to take in either 21 or 32 data, though gives no choice in cooling schedule, search mechanism or hyperparameters as the optimal configuration is used.

The Hybrid search is also implemented if *hybrid=True*.

The algorithm allows one to choose between *AIM* or *SDM* when implementing the function. The 21 unit case performs best under the following hyperparameters, as given by table 5.8 in the thesis:

$$\begin{aligned} case &: 21 - unit\ optimal\ implmentation \\ method\ for\ T_0 &: AIM \\ search\ algorithm &: Ejection\ Chain \\ cooling\ scheme &: Van\ Laarhoven \\ \delta &: 0.16 \\ max\ attempts &: 100n \end{aligned}$$

Similarly, the optimal hyperparameter configuration for the 32 case is given by:

$$\begin{aligned} case &: 32 - unit\ optimal\ implmentation \\ method\ for\ T_0 &: SDM \\ search\ algorithm &: Ejection\ Chain \\ cooling\ scheme &: Van\ Laarhoven \\ \delta &: 0.35 \\ max\ attempts &: 90n \end{aligned}$$

The function, however, only takes in a single starting solution, not a list as done elsewhere. The function below allows one to compute the optimal configuration for either the 21 or 32 unit case, as well as specify the relevant hyperparameters. The cooling schedule (Van Laarhoven) & move operator (Ejection Chain) are, however, fixed as they are what defines the optimal configuration as given by the paper (best implementation).

```
best_solution <- function(model_name, hybrid=TRUE, neighbourhood_size=NA, T_min=1, max_attempts=100*n,
                          AIM=TRUE, IEEE_RTS=FALSE, random_init, no_to_accept=12, no_to_attempt=100, del

  # ------ prints ------x
  print(paste('.................. Running Optimal Configuration:', model_name, '..................'))
  print(paste('.....................................................................'))
  main_start <- Sys.time()

  # ---- timer ----x
  start <- Sys.time()

  # ----  init solution ----x
  x <<- random_init
  y <<- compute_y_binary_matrix()

  # ----  init temperature ----x
  T0_ <- compute_init_temperature(x = x, x0 = 0.5, walk_length = 500)
  T0 <- ifelse(AIM,T0_$ave_inc,T0_$sd0)

  # --- init results ----x

  current <- x
  res <- compute_penalty(IEEE_RTS = IEEE_RTS); current_obj <- res$Q_total
```

```
not_accept_count <- 0
incumbent_sol <- current; incumbent_obj <- current_obj


# ------ store results ------x
solutions <- c()
temperature <- c()
total_accept <- c()
total_iters <- c()
Pens <- c()
Pws <- c()
Pls <- c()
Pcs <- c()
Pes <- c()
Qs <- c()
Q_totals <- c()
feasibility <- c()

Temp <- T0
timer <- 0


   # max_attempts = Omega Frozen
  while ((Temp>1) & (not_accept_count < max_attempts) & (timer<time_limit)) {
  # while (Temp>1) {
    print('.....................................')
    print(paste('Temperature: ', Temp))
    print(paste('Temperature Percentage: ', (Temp-1)/T0))
    print(paste('No Accept Count: ', not_accept_count))
    print('.....................................')
    print(paste('Proposed Score:', res$Q_total))
    print(paste('Current Score:', current_obj))
    print(paste('Incumbent Score:', incumbent_obj))
    print('                 ___                  ')

    no_accept <- 0
    no_attempt <- 0
    accepted <- FALSE

    while ((no_accept < no_to_accept*n) & (no_attempt < no_to_attempt*n)) {
    # while ((no_accept < 12*n) & (no_attempt < 100*n)) {
    # while ((no_attempt < 100)) {

      # time limit
      timer <- difftime(Sys.time(), start, units = 'hours')
      if (timer > time_limit) break

        # ------------------------------------------- Inner Loop: start ----------------------------
        no_attempt <- no_attempt + 1
        x <<- current; y <<- compute_y_binary_matrix()


        x <<- ejection_chain_operator(sol = x, el = el)
```

```
y <<- compute_y_binary_matrix()
res <- compute_penalty(IEEE_RTS = IEEE_RTS)




# ---- check acceptance criterion ----x
if (res$Q_total <= current_obj) {
  total_accept <- c(total_accept, 1)
  no_accept <- no_accept + 1
  accepted <- TRUE
  current_obj <- res$Q_total
  current <- x

  # ------ save results ------x
  solutions <- c(solutions, x)
  temperature <- c(temperature, Temp)
  total_iters <- total_iters + 1
  Pens <- c(Pens, res$P)
  Pws <- c(Pws, res$Pw)
  Pls <- c(Pls, res$Pl)
  Pcs <- c(Pcs, res$Pc)
  Pes <- c(Pes, res$Pe)
  Qs <- c(Qs, res$Q)
  Q_totals <- c(Q_totals, res$Q_total)
  feasibility <- c(feasibility, res$feasible)
  # ------ save results ------x


  if(current_obj <= incumbent_obj) {
    incumbent_obj <- current_obj
    incumbent_sol <- current

    # ----- local search -----x
    if (hybrid) {
      hyb <- local_search_hybrid(incumbent_sol = incumbent_sol, neighbourhood_size = neighbou
                                 search_operator=ejection_chain_operator, prints=FALSE)
      if (hyb$score < incumbent_obj) {
        incumbent_sol <- hyb$incumbent_sol
        incumbent_obj <- hyb$score
      }
    }
  }
} else {
  if (runif(1) < exp(-(res$Q_total - current_obj)/Temp)) {
    total_accept <- c(total_accept, 1)
    no_accept <- no_accept + 1
    current_obj <- res$Q_total
    current <- x
    accepted <- TRUE

    # ------ save results ------x
```

```
            solutions <- c(solutions, x)
            temperature <- c(temperature, Temp)
            total_iters <- total_iters + 1
            Pens <- c(Pens, res$P)
            Pws <- c(Pws, res$Pw)
            Pls <- c(Pls, res$Pl)
            Pcs <- c(Pcs, res$Pc)
            Pes <- c(Pes, res$Pe)
            Qs <- c(Qs, res$Q)
            Q_totals <- c(Q_totals, res$Q_total)
            feasibility <- c(feasibility, res$feasible)
            # ------ save results ------x

          }
          else total_accept <- c(total_accept, 0)

        }
      # ------------------------------------------- Inner Loop: finished ---------------------------
      }

      # ---- update sigma_s ----x
      sig_s <- sd(Q_totals)

      # ---- cumulative accept count < threshold ----x
      ifelse(accepted,
             not_accept_count <- 0,
             not_accept_count <- not_accept_count + 1)

      # ---- update temperature ----x
      Temp <- VanL(Temp, delta = delta, sig_s = sig_s)

      timer <- difftime(Sys.time(), start, units = 'hours')


  }

  end <- Sys.time()



  # ---- save results ----x
  fin <- list(model_name=model_name, search_operator=ejection_chain_operator,
              T_min=T_min, max_attempts=max_attempts, AIM=AIM, IEEE_RTS=IEEE_RTS, random_init=random_
              delta=delta,
              runtime=end-start, Pens=Pens, Pws=Pws, Pls=Pls, Pcs=Pcs, Pes=Pes, Qs=Qs, Q_totals=Q_tota
              incumbent_obj=incumbent_obj, incumbent_sol=incumbent_sol,
              temperature=temperature, total_accept=total_accept, feasibility=feasibility, solutions=


print(paste('.....................................................................'))
print(Sys.time()-main_start)
print(paste('.....................................................................'))
print(paste('.......................... Complete Run ............................'))
```

```
  return(fin)

}
```

21-Unit Instance

Here we implement the *21-unit* instance.

21-unit implementation:

We implement the algorithm in an identical fashion to that done in the paper, we cannot, however, compute as many permutations as the paper due to runtime constraints. The neighbourhood size ($n \times m$), minimum temperature ($T_{min} = 1$ & $no.\ accept = 12$ & $no.\ attempt = 100n$ are set as per the paper.

$\Omega_{frozen} = max\ attempts$, however, was significantly reduced due to runtime constraints. I attempted to set it at the paper specified figure of $100n$ however the algorithm did not converge within reasonable runtime. It also appeared to offer no significant improvement, because the implementation prints current scores, current incumbent objective function etc, it's easy to observe that no improvement is being made.

Alternatively we simple cut the run after a number of hours & report the results thus far.

As such, we set $\Omega_{frozen} = max\ attempts = 210$ (as apposed to the recommended $100n = 2100$). This simple allows for a more exhaustive search before declaring the algorithm is frozen, thus I do not think we prohibited the results greatly.

Here we run the implementation, again the results were pre-saved due to computational constraining the ability to compute on *knit time*.

```
load_data(FALSE)
el <- cbind(e,l)
n <- size
# best_21_unit_implementation_2 <- best_solution(model_name = 'Best 21-unit Implemenation', hybrid = TR
#                                                T_min = 1, random_init = inits_21[[1]], delta = 0.16,
#                                                 max_attempts = 210, #100*n, # Omega Frozen
#                                                no_to_accept = 12, no_to_attempt = 100, time_limit = 2)




# ---- save results ----x
# saveRDS(best_21_unit_implementation_2, file = "./final runs/best_21_unit_implementation_22.rds")

# ---- load results ----x
best_21_unit_implementation <- readRDS(file = "./final runs/best_21_unit_implementation_22.rds")
```

Now we can define two functions that compute the summary statistics of each run, to analyse results.

```
# ---- summary 1 ----x
best_run_summary <- function(run) {
  print(paste('Run name:', run$model_name))

  print('_____________________________________________________')
  print('Runtime:')
  print(run$runtime)
```

```
  print('_____________________________________________')
  print('available data:')
  print(names(run))
  print('_____________________________________________')
  print('Incumbent solution: ')
  print(run$incumbent_sol)

  print('_____________________________________________')
  print(paste('Incumbent score: ', round(run$incumbent_obj), sep=''))

}

# ---- summary 2 ----x
compute_best_summary <- function(res) {

  print('                          General Statistics                          ')
  print('______________________________________________________________________')
  print(paste('Best Objective: '))
  print(res$incumbent_sol)
  print('- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -')
  print(paste('Best Objective: ', res$incumbent_obj))
  print('- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -')
  print(paste('Proportion of Infeasible Solutions: ', 1-mean(unlist(res$feasibility))))
  print('- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -')
  print(res$runtime)
  print('- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -')


  print('                          Weighted Penalties                          ')
  print('______________________________________________________________________')

  print('Weights used: ')
  print(W)
  print('- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -')

  print('Overall Total Penalty: ')
  print(paste('average:', mean(res$Pens)))
  print(paste('minimum:', min(res$Pens)))
  print('- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -')

  print('Window Constraint: ')
  print(paste('average:', mean(res$Pws)))
  print(paste('minimum:', min(res$Pws)))
  print('- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -')

  print('Load Requirement Penalty: ')
  print(paste('average:', mean(res$Pls)))
  print(paste('minimum:', min(res$Pls)))
  print('- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -')

  print('Crew Constraint: ')
  print(paste('average:', mean(res$Pcs)))
  print(paste('minimum:', min(res$Pcs)))
```

```
  print('- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -')

  print('Exclusion Penalty: ')
  print(paste('average:', mean(res$Pes)))
  print(paste('minimum:', min(res$Pes)))
  print('- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -')




  # ----- Graph 1: Solution Results -----x
  plot(1:length(res$Q_totals), res$Q_totals, frame=F, 'l', col='steelblue', main=paste('Objective Functi
       ylab='Objective Score', xlab='Iterations', ylim = c(min(unlist(res$Qs))-100, max(unlist(res$Q_to
  lines(1:length(res$Q_totals), res$Qs,  'l', col='#800000')
  abline(h = min(unlist(res$Q_totals)), lty=2, col='darkblue')
  abline(h = res$incumbent_score, lty=2, col='darkblue')
  legend('topright', col = c('steelblue', '#800000', 'darkblue'),
         legend = c('with penalty (wp)', 'without penalty', 'Incumbent (wp)'), lty = c(1,1,2), box.lwd =



  # ----- Graph 2: Average Penalty -----x
  vals <- c(mean(unlist(res$Pens)), mean(unlist(res$Pws)), mean(unlist(res$Pls)), mean(unlist(res$Pcs))
  vals[is.na(vals)] <- 0
  names <- c('Total Penalty', 'Window Constraint', 'Load Penalty', 'Crew Constraint', 'Exclusion Penalty
  barplot(height = vals, col = sample(colors(), 5), legend.text = names, main= paste('Weighted Penalties

}
```

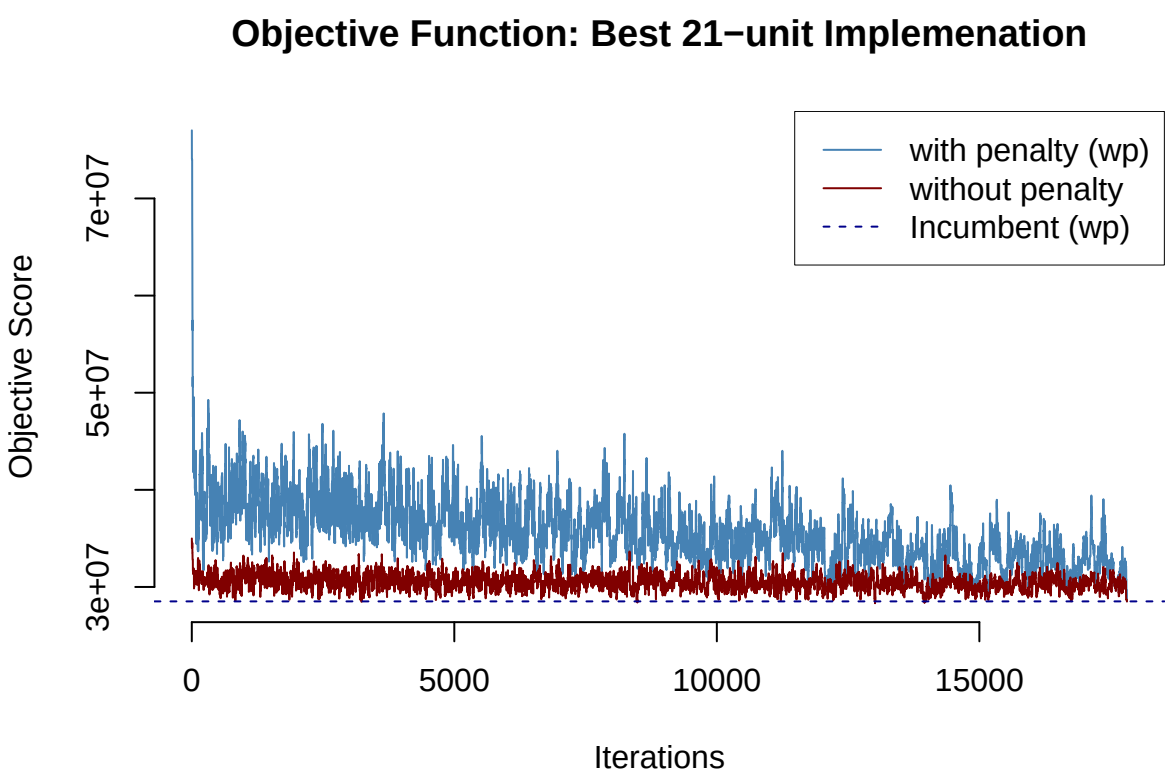
Here we call these functions on the best implementation of the 21 unit case:

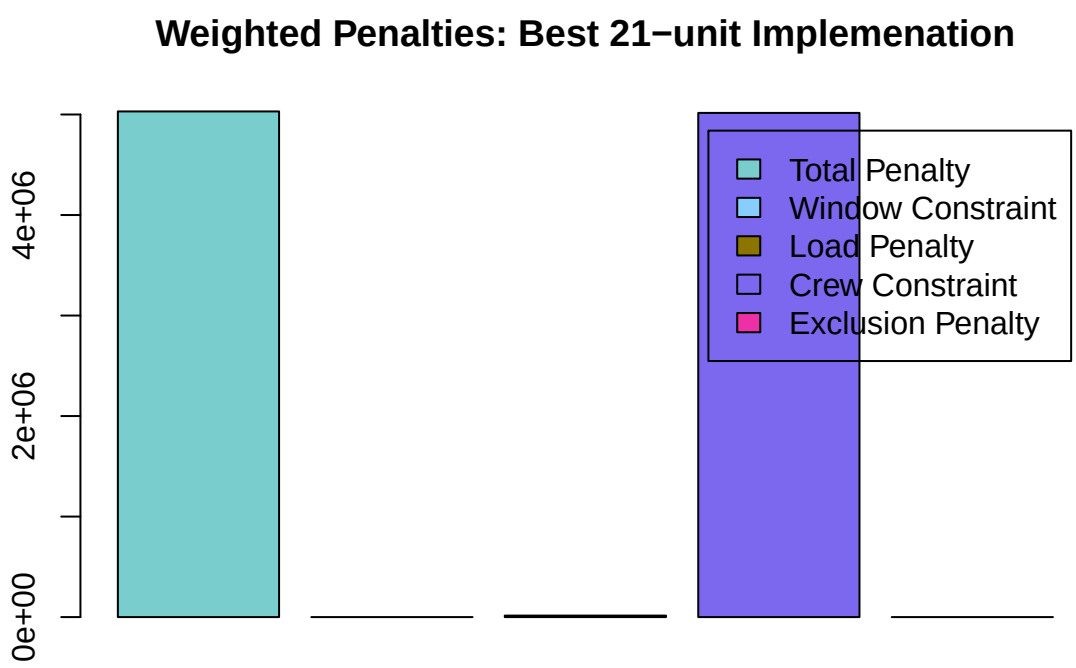
```
best_run_summary(best_21_unit_implementation)
```

```
## [1] "Run name: Best 21-unit Implemenation"
## [1] "_____________________________________________"
## [1] "Runtime:"
## Time difference of 33.92229 mins
## [1] "_____________________________________________"
## [1] "available data:"
##  [1] "model_name"      "search_operator" "T_min"           "max_attempts"
##  [5] "AIM"             "IEEE_RTS"        "random_init"     "delta"
##  [9] "runtime"         "Pens"            "Pws"             "Pls"
## [13] "Pcs"             "Pes"             "Qs"              "Q_totals"
## [17] "incumbent_obj"   "incumbent_sol"   "temperature"     "total_accept"
## [21] "feasibility"     "solutions"
## [1] "_____________________________________________"
## [1] "Incumbent solution: "
##  [1] 16 41 13 26 27 23  5 46  1 11 15 50 20 16  8  3 34 32 35 37 10
## [1] "_____________________________________________"
## [1] "Incumbent score: 28451311"
```

```
compute_best_summary(best_21_unit_implementation)
```

```
## [1] "                          General Statistics                          "
## [1] "______________________________________________________________________"
## [1] "Best Objective: "
##  [1] 16 41 13 26 27 23  5 46  1 11 15 50 20 16  8  3 34 32 35 37 10
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Best Objective:  28451311.045"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Proportion of Infeasible Solutions:  1"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## Time difference of 33.92229 mins
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "                          Weighted Penalties                          "
## [1] "______________________________________________________________________"
## [1] "Weights used: "
## [1] 5e+05 1e+00 2e+05 0e+00
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Overall Total Penalty: "
## [1] "average: 5030539.5977542"
## [1] "minimum: 13534"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Window Constraint: "
## [1] "average: 0"
## [1] "minimum: 0"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Load Requirement Penalty: "
## [1] "average: 14729.1435405087"
## [1] "minimum: 13425.85"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Crew Constraint: "
## [1] "average: 5015810.45421369"
## [1] "minimum: 0"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Exclusion Penalty: "
## [1] "average: 0"
## [1] "minimum: 0"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
```


Objective Function: Best 21−unit Implemenation
with penalty (wp)
without penalty
Incumbent (wp)
Objective Score
3e+07
5e+07
7e+07
0
5000
10000
15000
Iterations



Examine the Results: The algorithm ran for almost 2 hours before terminating. The search appears highly effective in moving towards better solutions, it was also able to move towards more feasible solutions.

After a 6 hours test run the algorithm reported:

1 "....."

1 "Temperature: 190195.564665084"

1 "Temperature Percentage: 0.19167698627902"

1 "No Accept Count: 0"

1 "....."

1 "Proposed Score: 41627959.9"

1 "Current Score: 28650192.755"

1 "Incumbent Score: 28151173.9"

So, slowly moving towards an optimal solution.

32-Unit Instance

```r
load_data(T)
el <- cbind(e,l)
# best_32_unit_implementation_2 <- best_solution(model_name = 'Best 32-unit Implemenation', hybrid = TR
#                                                T_min = 1, random_init = inits_32[[1]], delta = 0.35,
#                                                max_attempts = 500, #100*n, # Omega Frozen
#                                               no_to_accept = 12, no_to_attempt = 20, time_limit = 1)




# ---- save results ----x
# saveRDS(best_32_unit_implementation, file = "./final runs/best_31_unit_implementation_22.rds")

# ---- load results ----x
best_32_unit_implementation <- readRDS(file = "./final runs/best_32_unit_implementation_22.rds")
```

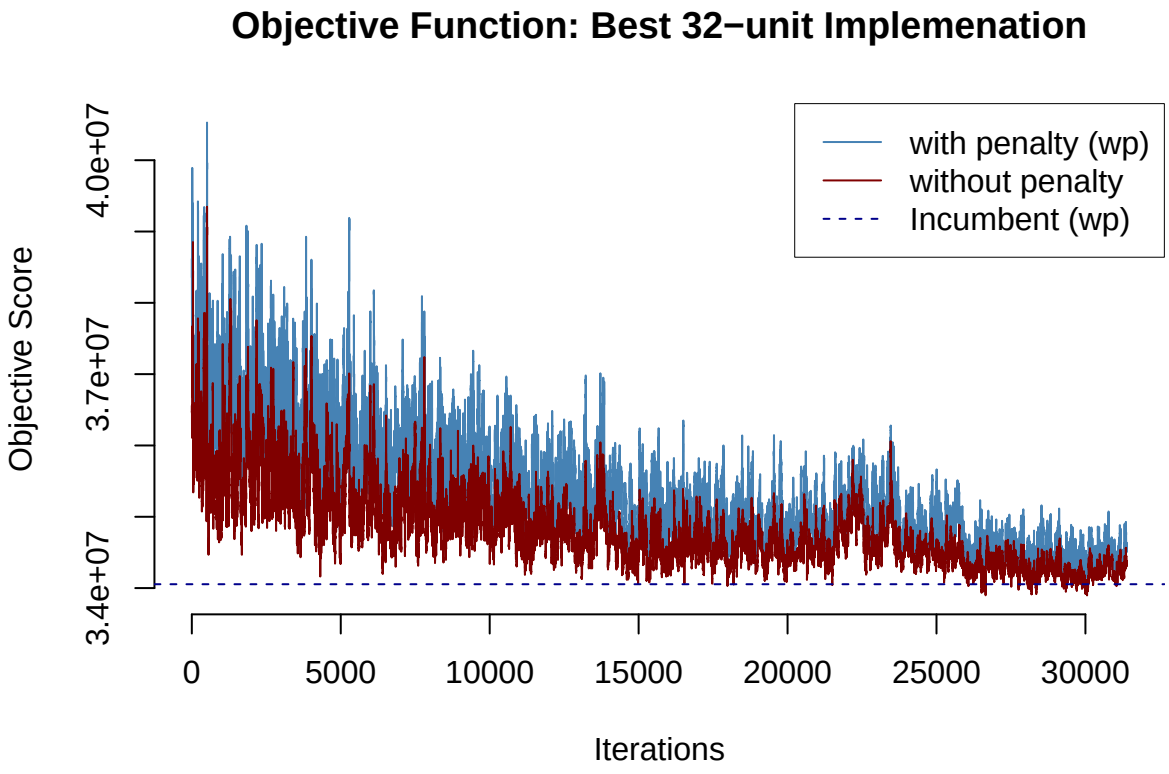
```r
best_run_summary(best_32_unit_implementation)
```

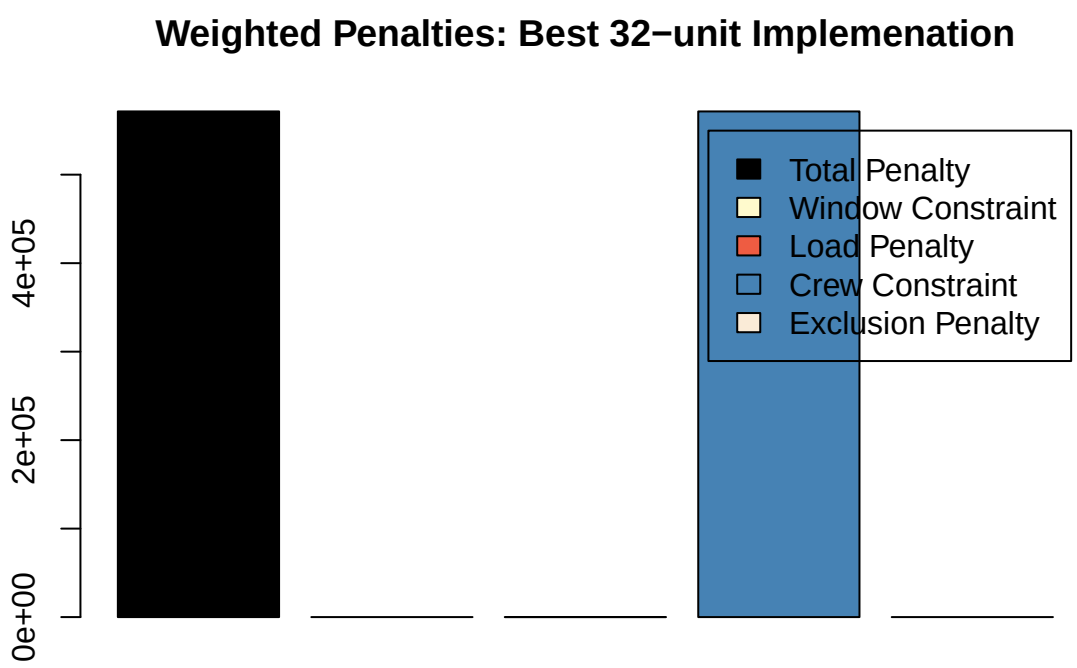
```
## [1] "Run name: Best 32-unit Implemenation"
## [1] "_____________________________________________"
## [1] "Runtime:"
## Time difference of 30.17551 mins
## [1] "_____________________________________________"
## [1] "available data:"
##  [1] "model_name"      "search_operator" "T_min"           "max_attempts"
##  [5] "AIM"             "IEEE_RTS"        "random_init"     "delta"
##  [9] "runtime"         "Pens"            "Pws"             "Pls"
## [13] "Pcs"             "Pes"             "Qs"              "Q_totals"
## [17] "incumbent_obj"   "incumbent_sol"   "temperature"     "total_accept"
## [21] "feasibility"     "solutions"
## [1] "_____________________________________________"
## [1] "Incumbent solution: "
##  [1] 21  7  4 34 15 43 22 36 45  1 29 14  9 41 19 43 47 20 41 16 27 10 31 32  8
## [26] 39 23 48  4  6 25 37
## [1] "_____________________________________________"
## [1] "Incumbent score: 34043266"
```

```r
compute_best_summary(best_32_unit_implementation)
```

```
## [1] "                         General Statistics                          "
## [1] "_____________________________________________________________________"
## [1] "Best Objective: "
##  [1] 21  7  4 34 15 43 22 36 45  1 29 14  9 41 19 43 47 20 41 16 27 10 31 32  8
## [26] 39 23 48  4  6 25 37
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Best Objective:  34043266"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Proportion of Infeasible Solutions:  0.998311779320889"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## Time difference of 30.17551 mins
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "                         Weighted Penalties                          "
```

```
## [1] "_____________________________________________________________________"
## [1] "Weights used: "
## [1] 40000     1 20000 20000
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Overall Total Penalty: "
## [1] "average: 571466.094318978"
## [1] "minimum: 0"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Window Constraint: "
## [1] "average: 0"
## [1] "minimum: 0"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Load Requirement Penalty: "
## [1] "average: 16.7727925718289"
## [1] "minimum: 0"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Crew Constraint: "
## [1] "average: 571449.321526406"
## [1] "minimum: 0"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Exclusion Penalty: "
## [1] "average: 0"
## [1] "minimum: 0"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
```





Very similar results to the 21 unit case are achieved, one could call into question the weightings of the penalty functions, as they majority of the penalty arises from *crew constraints* - which may or may not be a good/realistic assumption.

Simulated Annealing Implementation

Now that we have completed the assignment, we can also run a few of the other specifications out of interest. Again, the runtime was cut short due to computational constraints, however we can still investigate the implementation.

The code below allows one to specify any variant of the either the 21 or 32 unit implementations, after compiling we can try many variations.

The optimal parameterization was taken from table 5.8 for the simulated annealing implementations.

The same random initialized starting points are used, however again, we cannot run all 50 runs due to computational limits, so instead we will run the first of each specification.

Here is the pseudo-code:

Algorithm 4.7: The GMS simulated annealing algorithm

Input: A power system scenario for which to solve the generator maintenance scheduling problem

Output: The best maintenance schedule found

```
1  dataset ← declareSystemData()
2  [current, currentObj ] ← generateRandomSolution(dataset)
3  [avgT_0, stdT_0] ← initialTemperature(current, currentObj, dataset)
4  T ← avgT_0                                                    // or stdT_0
5  [incumbent, incumbentObj] ← [current, currentObj]
6  notAcceptCounter ← 0
7  while (T > T_min) and (notAcceptCounter < Ω_frozen) do
8      numberAccept ← 0
9      numberAttempt ← 0
10     accepted ← false
11     while (numberAccept < 12n) and (numberAttempt < 100n) do
12         numberAttempt ← numberAttempt + 1
13         neighbour ← current
14         unit ← rand([1, n])
15         chain ← createEjectionChainList(unit, n, e, ℓ, W_ext, neighbour)
16         Apply chain on neighbour to create new neighbour
17         P ← checkFeasibilityAndCalculatePenalty(neighbour, dataset)
18         Calculate neighbourObj
19         neighbourObj ← neighbourObj + P
20         ΔE ← neighbourObj − currentObj
21         if ΔE ≤ 0 then
22             [current, currentObj] ← [neighbour, neighbourObj]
23             numberAccept ← numberAccept + 1
24             accepted ← true
25             if currentObj < incumbentObj then
26                 [incumbent, incumbentObj] ← [current, currentObj]
27             end
28         else
29             if rand((0, 1)) < exp(ΔE/T) then
30                 [current, currentObj] ← [neighbour, neighbourObj]
31                 numberAccept ← numberAccept + 1
32                 accepted ← true
33             end
34         end
35     end
36     if accepted = true then
37         notAcceptCounter ← 0
38     else
39         notAcceptCounter ← notAcceptCounter + 1
40     end
41     Update temperature T
42 end
```

Note that the algorithm is identical for all cooling procedures, however different cooling schemes would require initializing different variables.

The function below should allow one to specify:

- The initial temperature technique technique (AIM or SDM)
- Cooling Method
- All hyper-parameters for a given cooling method
- Max Attempts

Although omitted from the table, $T_{min} = 1$ is used in all instances - as per the thesis.

The default parameters are those detailed as the best specification in the paper.

We wish to keep the functions universal in that it can implement any specification/instance of the problem. As such it must be able to take arguments for all/any cool schedule. which are all set to *NA* as default. The user would need to pass the cooling schedule of interest to the function.

The function also runs for the number of random inits that it is given. We only run one of each model variation, though one could readily run all 50 (as done in the paper). We simple cannot due to computational constraints. **as such: note that the algorithm takes a list of starting configurations, we simply pass a list of one. It also returns a list. Understanding this data structure is imperative**.

Note that the algorithm provided computes the random starting configuration in the function, however we allow the user to pass in a random solution set, as done in the random search, so that the same random configurations are used (as done in the papers results).

Passing Initial Solutions Note that initial solutions NEED to be passed as a list of solutions. Even if there is *1* solution is should be accessible as: *initList[[1]]*. This was a design decision to be able to compute all x runs in a single function call. The same is true of the abovementioned random search heuristic.

Compute σ_s adaptive parameter The paper describes: $\sigma_{\mathbf{s}}$ *is the stanardard deviation observed in the changing values* Thus all objective function values achieved are used, not only the accepted value, as it is the standard deviation observed in *changing* values not merely improvements.

The initial $\sigma_{\mathbf{s}}$ is computed directly by the *SDM* in providing T_0 - needed in some implementations such as Triki cooling.

Hyper parameters Hyper parameters are taken from the table - denoting the optimal configuration - unless otherwise stated.

Hyper parameters Some hyperparameters are given in the pseudocode & not the table, namely:

ADDITIONAL PARAMETERS:

- No. Accept : 12n
- No. Attempts : 100n

We will reduce these due to computational constraints

Here we define a function that, for any given list of starting configurations, returns a list of results of apply the specified SA algorithm to the data. Initialization criterion, cooling schedules, hyperparameters & model data are are variable, allowing for any required implementation.

```r
GMS_simulated_annealing <- function(model_name, search_operator, T_min=1, max_attempts=100*size, AIM=TRU
                                    no_to_accept=12, no_to_attempt=100,
                                    Geo_cool=FALSE, Huang_cool=FALSE, Van_Laarhoven_cool=FALSE, Triki_co
                                    alpha=NA, lambda=NA, delta=NA, zeta=NA, mu1=NA, mu2=NA,
                                    k1=2, k2=4, lambda1=2, time_limit) {


  # ------ prints ------x
  print(paste('.................. Running Computation:', model_name, '..................'))
  print(paste('.....................................................................'))
  main_start <- Sys.time()

  final_results <- list()
  for (init in 1:length(random_inits)) {

    print(paste('Run: ', init, '/', length(random_inits), sep = ''))
    print(paste('.....................................................................'))

    # ---- timer ----x
    start <- Sys.time()

    # ----  init solution ----x
    x <<- random_inits[[init]]
    y <<- compute_y_binary_matrix()

    # ----  init temperature ----x
    T0_ <- compute_init_temperature(x = x, x0 = 0.5, walk_length = 500)
    T0 <- ifelse(AIM,T0_$ave_inc,T0_$sd0)

    # --- init results ----x

    current <- x
    res <- compute_penalty(IEEE_RTS = IEEE_RTS); current_obj <- res$Q_total
    not_accept_count <- 0
    incumbent_sol <- current; incumbent_obj <- current_obj

    # ------ Triki Parameters -----x
    Delta <- T0_$sd0/mu2
    reinit <- TRUE
    doGeometric <- FALSE
    previous_ave_cost <- res$Q_total
    equib_not_reached <- 0
    negative_temp <- 0



    # ------ store results ------x
    solutions <- c()
    temperature <- c()
    total_accept <- c()
    total_iters <- c()
    Pens <- c()
    Pws <- c()
```

```
Pls <- c()
Pcs <- c()
Pes <- c()
Qs <- c()
Q_totals <- c()
feasibility <- c()

timer <- 0
Temp <- T0

# max_attempts = Omega Frozen

while (Temp>1 & not_accept_count < max_attempts & timer<time_limit) {
# while (Temp>1) {
  print('.....................................')
  print(paste('Temperature: ', Temp))
  print(paste('Temperature Percentage: ', (Temp-1)/T0))

  no_accept <- 0
  no_attempt <- 0
  accepted <- FALSE



  while ((no_accept < no_to_accept*n) & (no_attempt < (no_to_attempt*n))) {
  # while ((no_accept < 12*n) & (no_attempt < 100*n)) {
  # while ((no_attempt < 100)) {


      # ------------------------------------------ Inner Loop: start -----------------------------
      no_attempt <- no_attempt + 1
      x <<- current; y <<- compute_y_binary_matrix()
      x <<- search_operator(sol = x, el = el)
      y <<- compute_y_binary_matrix()
      res <- compute_penalty(IEEE_RTS = IEEE_RTS)

    #  print('.....................................')
    #  print(paste('Proposed Score:', res$Q_total))
    #  print(paste('Current Score:', current_obj))
    #  print(paste('Incumbent Score:', incumbent_obj))
    #  print('.....................................')

      # ---- check acceptance criterion ----x
      if (res$Q_total <= current_obj) {
        total_accept <- c(total_accept, 1)
        no_accept <- no_accept + 1
        accepted <- TRUE
        current_obj <- res$Q_total
        current <- x

        # ------ save results ------x
        solutions <- c(solutions, x)
        temperature <- c(temperature, Temp)
```

```
      total_iters <- total_iters + 1
      Pens <- c(Pens, res$P)
      Pws <- c(Pws, res$Pw)
      Pls <- c(Pls, res$Pl)
      Pcs <- c(Pcs, res$Pc)
      Pes <- c(Pes, res$Pe)
      Qs <- c(Qs, res$Q)
      Q_totals <- c(Q_totals, res$Q_total)
      feasibility <- c(feasibility, res$feasible)
      # ------ save results ------x


      if(current_obj <= incumbent_obj) {
        incumbent_obj <- current_obj
        incumbent_sol <- current
      }
    } else {
      if (runif(1) < exp(-(res$Q_total - current_obj)/Temp)) {
        total_accept <- c(total_accept, 1)
        no_accept <- no_accept + 1
        current_obj <- res$Q_total
        current <- x
        accepted <- TRUE

        # ------ save results ------x
        solutions <- c(solutions, x)
        temperature <- c(temperature, Temp)
        total_iters <- total_iters + 1
        Pens <- c(Pens, res$P)
        Pws <- c(Pws, res$Pw)
        Pls <- c(Pls, res$Pl)
        Pcs <- c(Pcs, res$Pc)
        Pes <- c(Pes, res$Pe)
        Qs <- c(Qs, res$Q)
        Q_totals <- c(Q_totals, res$Q_total)
        feasibility <- c(feasibility, res$feasible)
        # ------ save results ------x

      }
      else {
      total_accept <- c(total_accept, 0)
    }
    }
  # ------------------------------------------- Inner Loop: finished ---------------------------
  }

  # ---- update sigma_s ----x
  sig_s <- sigma_s <- sd(Q_totals)

  # ---- cumulative accept count < threshold ----x
  ifelse(accepted,
         not_accept_count <- 0,
         not_accept_count <- not_accept_count + 1)
```

```
# ---- update temperature ----x
if (Geo_cool) Temp <- Geo(alpha, Temp)

if (Huang_cool) Temp <- Huang(Temp, lambda = lambda, sig_s = sig_s)

if (Van_Laarhoven_cool) Temp <- VanL(Temp, delta = delta, sig_s = sig_s)

# ---- Triki Additional Computation ----x
if(Triki_cool) {


  current_ave_cost <- mean(Q_totals)
  # ---- as taken from Algorithm B.2 In the Thesis Appendix ----x
  if (!doGeometric) {

    if (!reinit) {

      equib_not_reached <- 0
      if ((current_ave_cost/(previous_ave_cost-Delta)) > zeta) {
          equib_not_reached <- equib_not_reached + 1
      } else {
          equib_not_reached <- 0
          }
    }
    if (equib_not_reached > k1) {

      reinit <- TRUE
      equib_not_reached <- 0
      Temp <- Geo(lambda1, Temp)
      Delta <- sigma_s/mu1
    } else if (((Temp*Delta)/sigma_s)>1) {

      negative_temp <- negative_temp + 1
      reinit <- TRUE
      if (negative_temp < k2) {

        Temp <- Geo(lambda1, Temp)
        Delta <- sigma_s/mu1

      } else {

          doGeometric <- TRUE
        }
    } else {

      reinit <- FALSE
      previous_ave_cost <- current_ave_cost
      Temp <- Triki(Ts = Temp, Delta = Delta, sig_s = sig_s)
    }

  } else {

    Temp <- Geo(lambda1, Temp)
```

```
          }
    }


        # set time diff
        timer <- difftime(Sys.time(), start, units = 'hours')
        #if (timer > 4) break
    }

    end <- Sys.time()

    # ---- save results ----x
    fin <- list(model_name=model_name, search_operator=search_operator,
                T_min=T_min, max_attempts=max_attempts, AIM=AIM, IEEE_RTS=IEEE_RTS, random_init=random_
                Geo_cool=Geo_cool, Huang_cool=Huang_cool, Van_Laarhoven_cool=Van_Laarhoven_cool, Triki_
                alpha=alpha, lambda=lambda, delta=delta, zeta=zeta, mu1=mu1, mu2=mu2,
                runtime=end-start, Pens=Pens, Pws=Pws, Pls=Pls, Pcs=Pcs, Pes=Pes, Qs=Qs, Q_totals=Q_tota
                incumbent_obj=incumbent_obj <- current_obj, incumbent_sol=incumbent_sol,
                temperature=temperature, total_accept=total_accept, feasibility=feasibility, solutions=s

    # ---- results for each run ----x
    final_results[[init]] <- fin

  }



  print(paste('.................................................................'))
  print(Sys.time()-main_start)
  print(paste('.................................................................'))
  print(paste('......................... Complete Run ..........................'))

  return(final_results)
}
```

Again, the results were saved & reloaded during knit time.

21-Unit : Geometric Cooling : Classical Search

Here we implment simulated annealing, under classical search, using geometric cooling. The paper details:

$$T0 = AIM$$
$$alpha = 0.95$$
$$max_attemps = 100n$$

```
# ---- load data ----x
el <- cbind(e,l)
n <- size
load_data(FALSE)
```

```r
# ---- run model ----x
# SA_21_classical_geo <- GMS_simulated_annealing(model_name = '21-Unit --- Geometric Cooling --- Classi
#                                                search_operator = classical_operator, max_attempts = 1
#                                                no_to_accept = 12, no_to_attempt = 100,
#                                               random_inits = inits_21[1], Geo_cool = TRUE, alpha = 0.


# ---- save results ----x
# saveRDS(SA_21_classical_geo, file = "./final runs/SA_21_classical_geo.rds")

# ---- load results ----x
SA_21_classical_geo <- readRDS(file = "./final runs/SA_21_classical_geo.rds")
```

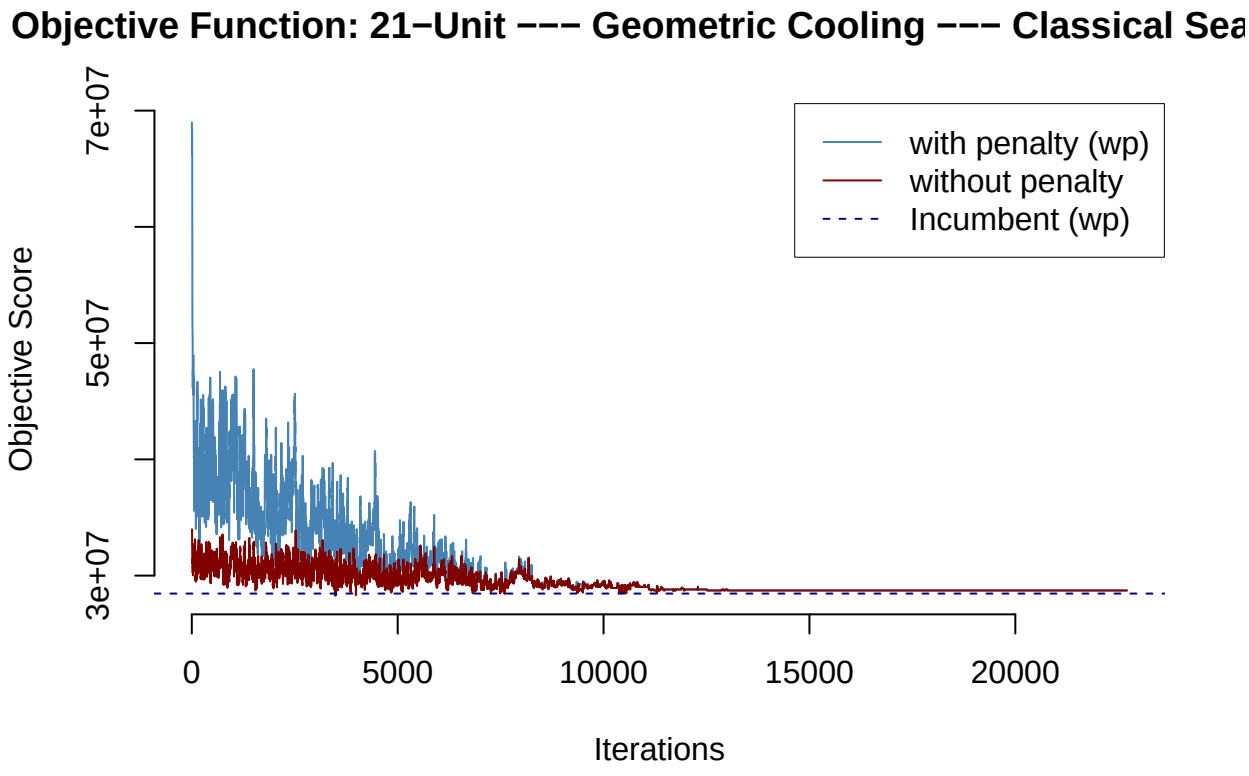
```r
best_run_summary(SA_21_classical_geo[[1]])
```

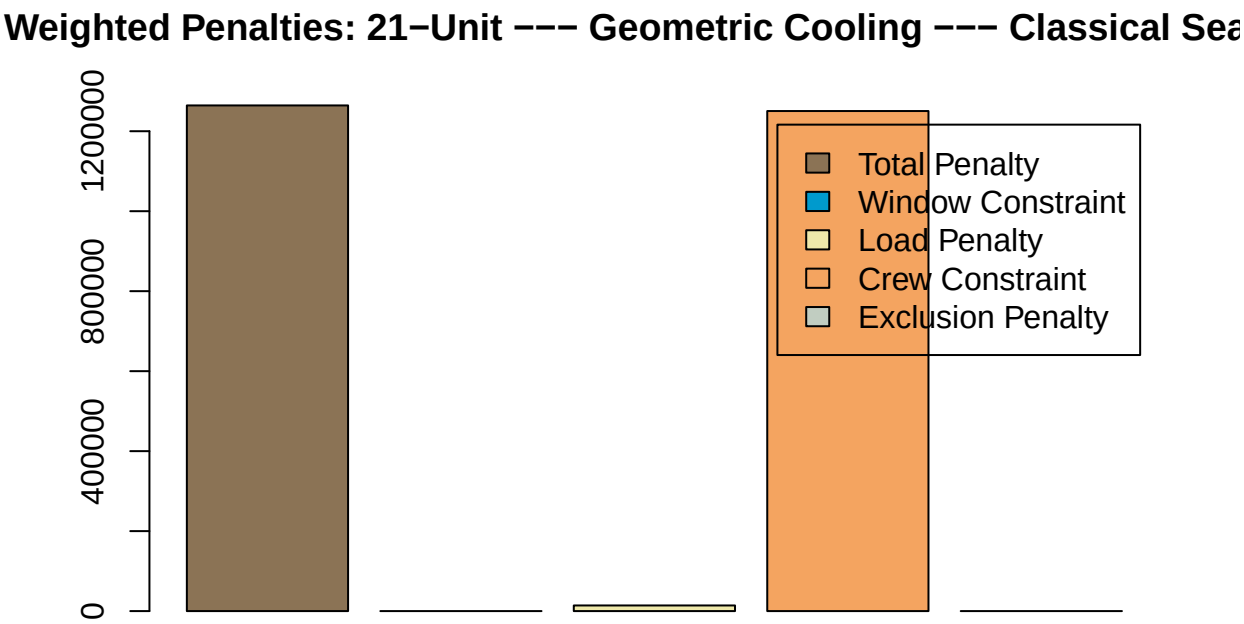
```
## [1] "Run name: 21-Unit --- Geometric Cooling --- Classical Search"
## [1] "_____________________________________________"
## [1] "Runtime:"
## Time difference of 2.003821 hours
## [1] "_____________________________________________"
## [1] "available data:"
##  [1] "model_name"         "search_operator"    "T_min"
##  [4] "max_attempts"       "AIM"                "IEEE_RTS"
##  [7] "random_init"        "Geo_cool"           "Huang_cool"
## [10] "Van_Laarhoven_cool" "Triki_cool"         "alpha"
## [13] "lambda"             "delta"              "zeta"
## [16] "mu1"                "mu2"                "runtime"
## [19] "Pens"               "Pws"                "Pls"
## [22] "Pcs"                "Pes"                "Qs"
## [25] "Q_totals"           "incumbent_obj"      "incumbent_sol"
## [28] "temperature"        "total_accept"       "feasibility"
## [31] "solutions"
## [1] "_____________________________________________"
## [1] "Incumbent solution: "
##  [1]  8 42 18  2 33 15 20 47 17  3  1 37 10 23 13  6 32 40 31 27  3
## [1] "_____________________________________________"
## [1] "Incumbent score: 28727705"
```

```r
compute_best_summary(SA_21_classical_geo[[1]])
```

```
## [1] "                        General Statistics                         "
## [1] "___________________________________________________________________"
## [1] "Best Objective: "
##  [1]  8 42 18  2 33 15 20 47 17  3  1 37 10 23 13  6 32 40 31 27  3
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Best Objective:  28727705.3275"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Proportion of Infeasible Solutions:  1"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## Time difference of 2.003821 hours
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "                        Weighted Penalties                         "
```

```
## [1] "_____________________________________________________________________"
## [1] "Weights used: "
## [1] 5e+05 1e+00 2e+05 0e+00
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Overall Total Penalty: "
## [1] "average: 1264475.2497248"
## [1] "minimum: 13510"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Window Constraint: "
## [1] "average: 0"
## [1] "minimum: 0"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Load Requirement Penalty: "
## [1] "average: 14050.3454933732"
## [1] "minimum: 13384"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Crew Constraint: "
## [1] "average: 1250424.90423143"
## [1] "minimum: 0"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Exclusion Penalty: "
## [1] "average: 0"
## [1] "minimum: 0"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
```





A local mimina is reached after approx 10'000 iterations, the performance isn't far off the best implementation.

21-Unit : HUANG Cooling : Classical Search

```
# ---- run model ----x
# SA_21_classical_huang <- GMS_simulated_annealing(model_name = '21-Unit --- Huang Cooling --- Classica
#                                                  search_operator = classical_operator, max_attempts =
#                                                  no_to_accept = 12, no_to_attempt = 100, Huang_cool =
#                                                random_inits = inits_21[1])

# ---- save results ----x
# saveRDS(SA_21_classical_huang, file = "./final runs/SA_21_classical_huang.rds")

# ---- load results ----x
SA_21_classical_huang <- readRDS(file = "./final runs/SA_21_classical_huang.rds")
```

```
best_run_summary(SA_21_classical_huang[[1]])
```

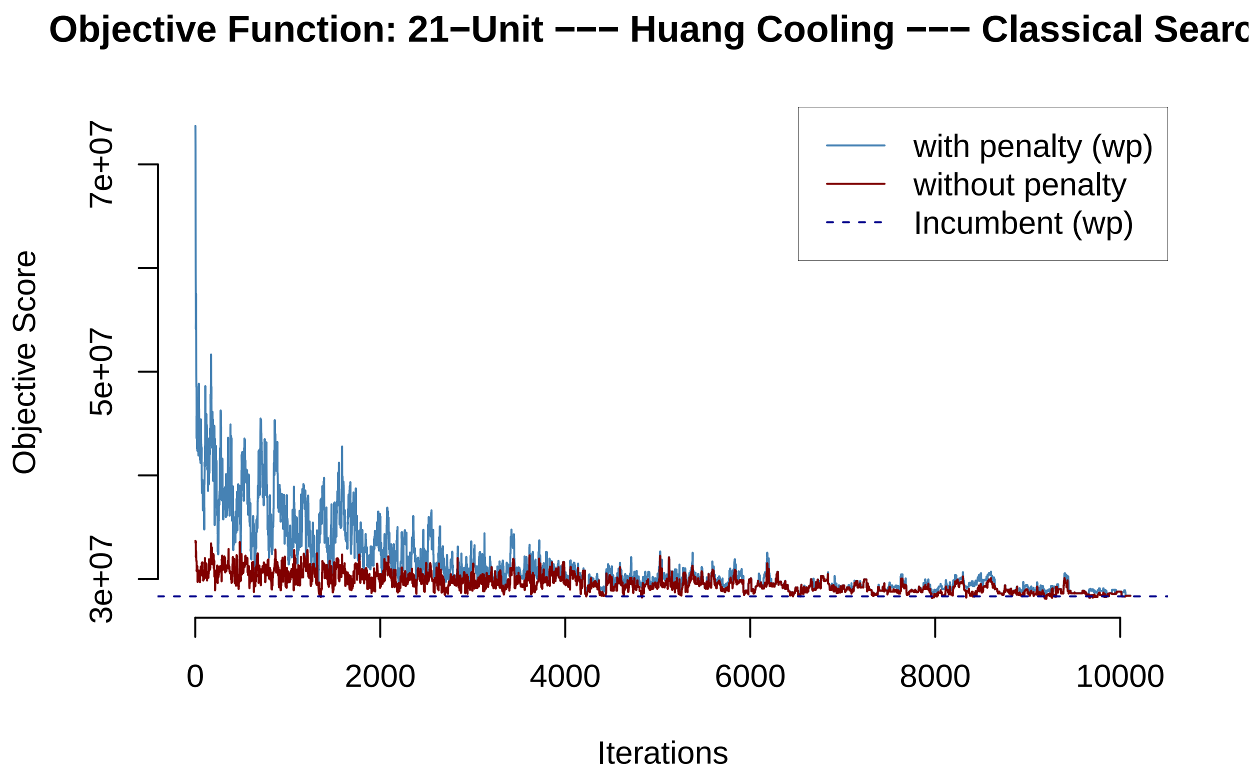
```
## [1] "Run name: 21-Unit --- Huang Cooling --- Classical Search"
## [1] "_____________________________________________"
## [1] "Runtime:"
```

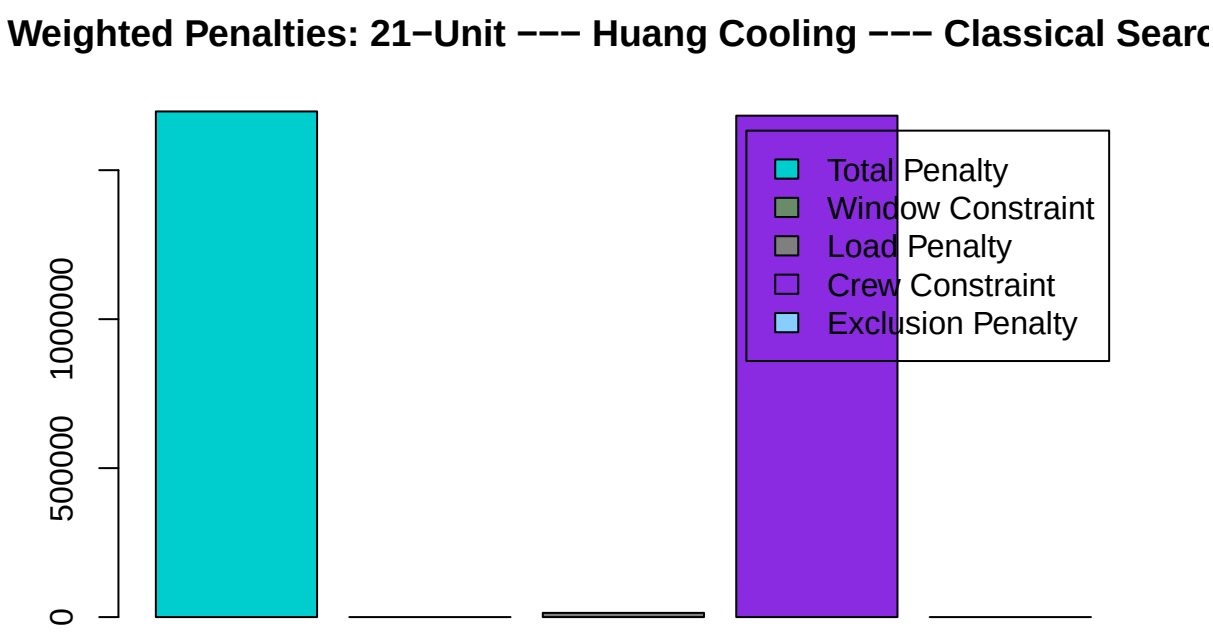
```
## Time difference of 30.2639 mins
## [1] "_____________________________________________"
## [1] "available data:"
##  [1] "model_name"         "search_operator"    "T_min"
##  [4] "max_attempts"       "AIM"                "IEEE_RTS"
##  [7] "random_init"        "Geo_cool"           "Huang_cool"
## [10] "Van_Laarhoven_cool" "Triki_cool"         "alpha"
## [13] "lambda"             "delta"              "zeta"
## [16] "mu1"                "mu2"                "runtime"
## [19] "Pens"               "Pws"                "Pls"
## [22] "Pcs"                "Pes"                "Qs"
## [25] "Q_totals"           "incumbent_obj"      "incumbent_sol"
## [28] "temperature"        "total_accept"       "feasibility"
## [31] "solutions"
## [1] "_____________________________________________"
## [1] "Incumbent solution: "
##  [1]  5 31 10  1 48 15 12 42 16  2 26 27 21  6 18 24 30 35 37 38  2
## [1] "_____________________________________________"
## [1] "Incumbent score: 28400402"
```

```
compute_best_summary(SA_21_classical_huang[[1]])
```

```
## [1] "                         General Statistics                         "
## [1] "____________________________________________________________________"
## [1] "Best Objective: "
##  [1]  5 31 10  1 48 15 12 42 16  2 26 27 21  6 18 24 30 35 37 38  2
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Best Objective:  28400401.9"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Proportion of Infeasible Solutions:  1"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## Time difference of 30.2639 mins
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "                         Weighted Penalties                         "
## [1] "____________________________________________________________________"
## [1] "Weights used: "
## [1] 5e+05 1e+00 2e+05 0e+00
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Overall Total Penalty: "
## [1] "average: 1697195.49091583"
## [1] "minimum: 13431.85"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Window Constraint: "
## [1] "average: 0"
## [1] "minimum: 0"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Load Requirement Penalty: "
## [1] "average: 14176.9962071012"
## [1] "minimum: 13255.7"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Crew Constraint: "
## [1] "average: 1683018.49470873"
## [1] "minimum: 0"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
```

```
## [1] "Exclusion Penalty: "
## [1] "average: 0"
## [1] "minimum: 0"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
```





Interesting to note how the search moves to more feasible solutions. This solution seems to converge slower than geo cooling.

32-Unit : Triki Cooling : Ejection Chain Search

Finanlly, we examine the Triki instance - applying to to the 32 unit dataset & utilizing the ejection chain cooling.

The Triki implementation is given in the appendix in the thesis:

Algorithm B.2: Simulated annealing with targeted average decrease in cost

```
1  dataset ← declareSystemData()
2  [current, currentObj ] ← generateRandomSolution(dataset)
3  [avgT_0, stdT_0] ← initialTemperature(current, currentObj, dataset)
4  T ← stdT_0
5  σ ← stdT_0
6  Δ ← σ/μ_2
7  negativeTemperature ← 0
8  reinitialising ← true
9  doGeometric ← false
10 while system not frozen do
11     Do the inner loop of SA and return the standard deviation     // Metropolis loop
12     σ ← standard deviation
13     if doGeometric = false then
14         if reinitialising = false then
15             if currentAverageCost/(previousAverageCost − Δ) > ζ then
16                 equilibriumNotReached ← equilibriumNotReached + 1
17             else
18                 equilibriumNotReached ← 0
19             end
20         end
21         if equilibriumNotReached > K_1 then
22             reinitialising ← true
23             equilibriumNotReached ← 0
24             T ← geometricCooling(λ_1, T)                    // reheating (λ_1 > 1)
25             Δ ← σ/μ_1
26         else if TΔ/σ^2 > 1 then
27             negativeTemperature ← negativeTemperature + 1
28             reinitialising ← true
29             if negativeTemperature < K_2 then
30                 T ← geometricCooling(λ_1, T)                // reheating (λ_1 > 1)
31                 Δ ← σ/μ_1
32             else
33                 doGeometric ← true
34             end
35         else
36             reinitialising ← false
37             previousAverageCost ← currentAverageCost
38             T ← TrikiCooling(Δ, σ, T)
39         end
40     else
41         T ← geometricCooling(λ_2, T)
42     end
43 end
```

As abovementioned the triki algorithm requires additional hyperparameters (specified above) as well has

offers a *heating* (as apposed to cooling) mechanism that allows the algorithm to 'gain' energy.

```
# ---- load data ----x
load_data(TRUE)
el <- cbind(e,l)
n <- size

# SA_32_ejection_triki <- GMS_simulated_annealing(model_name = '32-Unit --- Triki Cooling --- Ejection
#                                                 search_operator = ejection_chain_operator, max_attemp
#                                                 no_to_accept = 12, no_to_attempt = 100, Triki_cool =
#                                                 random_inits = inits_32[1], mu1=10, mu2=10, zeta=1.06
#                                                 k1=2, k2=4, lambda1=2, time_limit=0.5)


# ---- save results ----x
# saveRDS(SA_32_ejection_triki, file = "./final runs/SA_32_ejection_triki.rds")

# ---- load results ----x
SA_32_ejection_triki <- readRDS(file = "./final runs/SA_32_ejection_triki.rds")
```

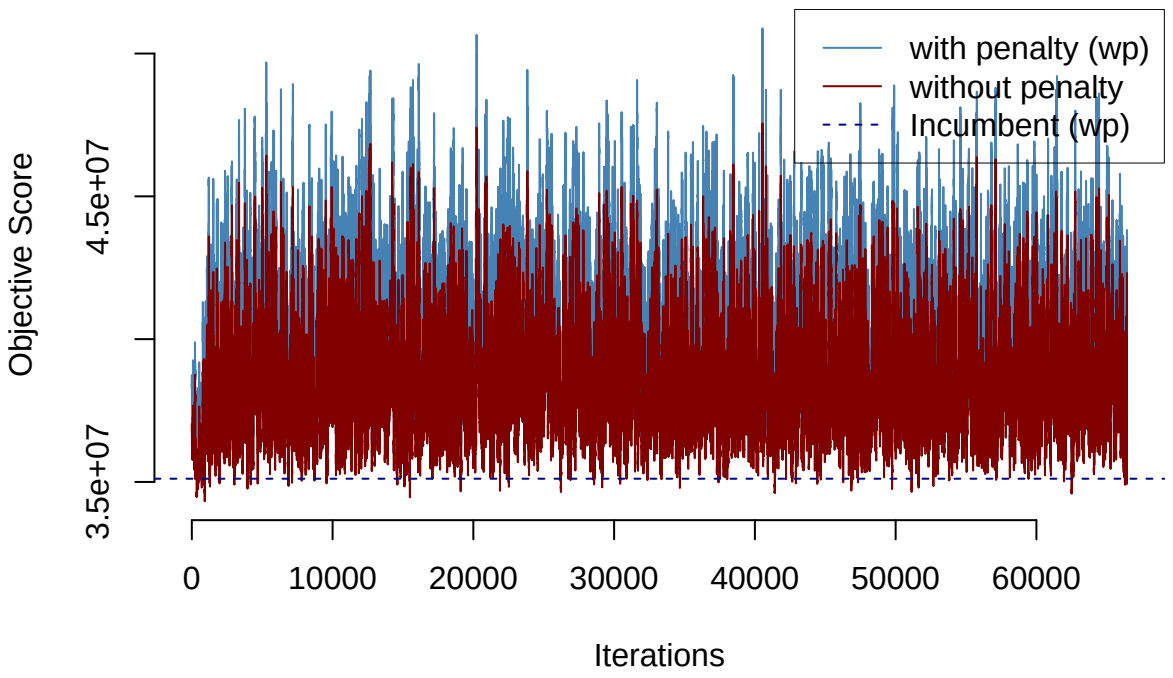
```
best_run_summary(SA_32_ejection_triki[[1]])
```

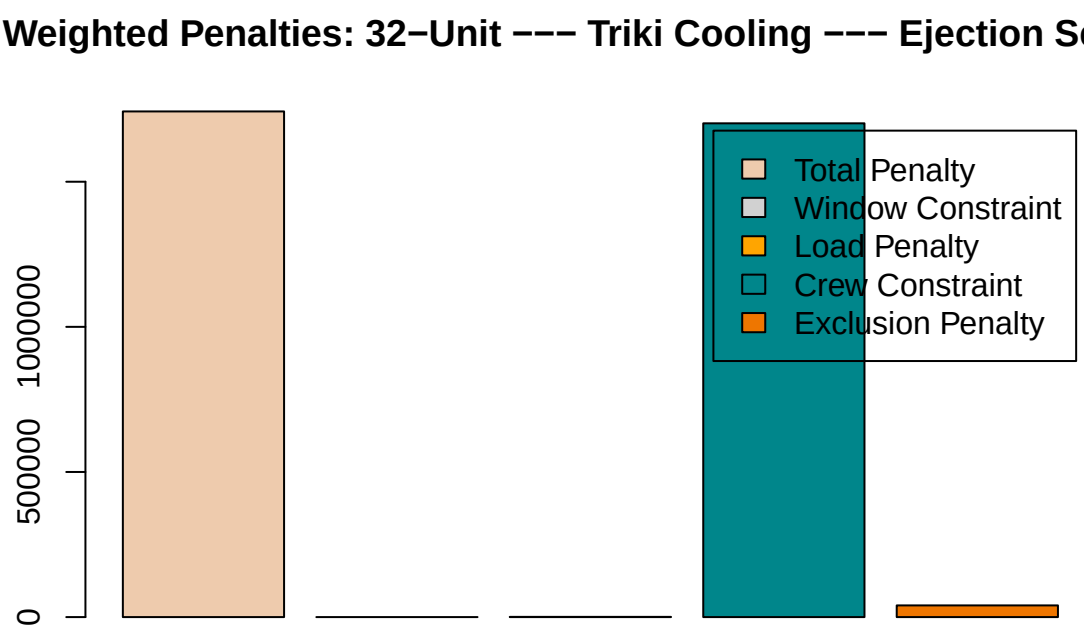
```
## [1] "Run name: 32-Unit --- Triki Cooling --- Ejection Search"
## [1] "_____________________________________________"
## [1] "Runtime:"
## Time difference of 30.19501 mins
## [1] "_____________________________________________"
## [1] "available data:"
##  [1] "model_name"         "search_operator"    "T_min"
##  [4] "max_attempts"       "AIM"                "IEEE_RTS"
##  [7] "random_init"        "Geo_cool"           "Huang_cool"
## [10] "Van_Laarhoven_cool" "Triki_cool"         "alpha"
## [13] "lambda"             "delta"              "zeta"
## [16] "mu1"                "mu2"                "runtime"
## [19] "Pens"               "Pws"                "Pls"
## [22] "Pcs"                "Pes"                "Qs"
## [25] "Q_totals"           "incumbent_obj"      "incumbent_sol"
## [28] "temperature"        "total_accept"       "feasibility"
## [31] "solutions"
## [1] "_____________________________________________"
## [1] "Incumbent solution: "
##  [1] 16 11  6 44  7 43 14 47 48 32 41 12 13 39 29 17  3  4 34  1 31  9 35 26 36
## [26] 18 42 26 51 20  6 27
## [1] "_____________________________________________"
## [1] "Incumbent score: 43821720"
```

```
compute_best_summary(SA_32_ejection_triki[[1]])
```

```
## [1] "                          General Statistics                          "
## [1] "______________________________________________________________________"
## [1] "Best Objective: "
##  [1] 16 11  6 44  7 43 14 47 48 32 41 12 13 39 29 17  3  4 34  1 31  9 35 26 36
```

```
## [26] 18 42 26 51 20  6 27
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Best Objective:  43821719.7525"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Proportion of Infeasible Solutions:  1"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## Time difference of 30.19501 mins
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "                          Weighted Penalties                           "
## [1] "_______________________________________________________________________"
## [1] "Weights used: "
## [1] 40000     1 20000 20000
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Overall Total Penalty: "
## [1] "average: 1742203.53887283"
## [1] "minimum: 40178.9"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Window Constraint: "
## [1] "average: 0"
## [1] "minimum: 0"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Load Requirement Penalty: "
## [1] "average: 739.485404624276"
## [1] "minimum: 0"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Crew Constraint: "
## [1] "average: 1701184.06791908"
## [1] "minimum: 40000"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
## [1] "Exclusion Penalty: "
## [1] "average: 40279.985549133"
## [1] "minimum: 0"
## [1] "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -"
```


Objective Function: 32−Unit −−− Triki Cooling −−− Ejection Search
with penalty (wp)
without penalty
Incumbent (wp)
Objective Score
4.5e+07
3.5e+07
0
10000
20000
30000
40000
50000
60000
Iterations



The Triki implementation appears far more noisy, less precise & does not appear to improve effeciently.

Conclusion

All the runs appear similar, though the functional setup allows one to examine/explore any configuration. The implementations was a very interested, real world application of meta-heuristics.

Finally, we also run various configurations, testing every cooling scheme, dataset & search operator. The code is modular allowing users to design any configuration that the see fit, as well use offers many concise, summary statistics functions to analyse results.
