CoBrInUS VRP

Team Members: Anmol Pahwa, Harshvardhan Shivram Ket, Miguel Jaller, Ruben Yie Pinedo, Claudio Barbieri da Cunha, Marcos Roberto Silva

Framework:

The Adaptive Large Neighborhood Search (ALNS) metaheuristic *searches* through the *neighborhood* by destroying and consequently rebuilding the solution thereby reconfiguring *large* portions of the solution using specific operators that are chosen *adaptively* in each iteration of the algorithm based on the performance of operators in the previous iterations, hence the name adaptive large neighborhood search (Ropke and Pisinger, 2006). Below is a detailed description of the framework of this algorithm along with its implementation (and validation), tailored towards generating high-quality solutions for the synthetic distribution environments described in the previous section.

The implementation in this work follows the standard ALNS structure. It maintains a current solution – s, and tracks the best solution found so far – s^*, initializing both from a starting configuration – s_o. The core search proceeds iteratively over j segments, each comprising n individual iterations. The fundamental operation within each iteration is a ruin-and-recreate cycle. This begins with the selection of a removal and an insertion operator – o_r and o_i, from predefined sets $\boldsymbol{o_r}$ and $\boldsymbol{o_i}$, respectively. This selection is guided by a roulette wheel mechanism employing adaptive weights that reflect the recent historical performance of each operator ($w_r; o_r \in \boldsymbol{o_r}$ and $w_i; o_i \in \boldsymbol{o_i}$). The chosen removal operator then partially destroys the current solution, removing a certain quantity of elements (ranging between $\underline{e}$ and $\overline{e}$) or fraction of elements (ranging between $\underline{\mu}$ and $\overline{\mu}$). Following the ruin phase, the selected insertion operator rebuilds the solution, generating a new candidate solution s'.

A key aspect of the ALNS methodology is its adaptive learning mechanism for operator selection. The algorithm monitors operator success within each segment using scores ($\pi_r; o_r \in \boldsymbol{o_r}$ and $\pi_i; o_i \in \boldsymbol{o_i}$), which are reset at the start of every segment. In each iteration, the operator scores are updated based on the quality and uniqueness of the solutions generated. Subsequently, at the conclusion of a segment, the weights w_r and w_i are updated based on these accumulated scores, operator usage counts ($c_r; o_r \in \boldsymbol{o_r}$ and $c_i; o_i \in \boldsymbol{o_i}$), and a reaction factor – ρ. This factor determines how strongly recent performance influences the weights, while a dissipation factor of $(1-\rho)$ provides stability by retaining some influence from previous segments. The recalculated weights then guide operator selection in the subsequent segment.

The acceptance of the newly generated solution s' into the search process depends on its objective function value f relative to the current solution s and the best-known solution s^*, potentially considering solution novelty determined via a hash function h. If s' yields a better objective value than s^* ($f(s') < f(s^*)$), it is unconditionally accepted, replacing both s and s^*, and the operators used receive the highest reward, amounting to σ_1. If s' improves only upon the current solution ($f(s^*) \leq f(s') < f(s)$), it replaces s, and the operators earn a standard reward of σ_2. To facilitate escape from local optima, the framework also incorporates a mechanism, akin to Simulated Annealing, to accept non-improving solutions ($f(s') > f(s)$). Such solutions are accepted probabilistically based on the Boltzmann criterion, $\lambda < \exp(f(s) - f(s')/T_k)$(where $\lambda \in [0,1]$ is random), with acceptance more likely at higher temperatures – T_k. Operators leading to such accepted solutions are given a smaller reward of σ_3.

The algorithm initializes this temperature at a level designed to permit acceptance of an $\overline{\omega}$ worse solution, with a target probability $\overline{\tau}$. In each subsequent iteration, the temperature is reduced by a cooling factor φ, progressively decreasing the likelihood of accepting inferior solutions, to a minimum that would still enable the algorithm to accept an $\underline{\omega}$ worse solution with a $\underline{\tau}$ probability. To prevent stagnation, the search is periodically refocused by resetting the current solution s to the best-found solution s^* every k segments. Additionally, solution quality is further refined through an optional local search phase applied at the end of each segment, running for m iterations using operators from a set $\boldsymbol{o_l}$. After completing the predefined n segments, the ALNS procedure terminates, returning the best solution s^* identified.

Algorithm Adaptive Large Neighbourhood Search (ALNS)

Procedure ALNS $\left(s_o, \left(j, k, n, m, \boldsymbol{o_r}, \boldsymbol{o_i}, \boldsymbol{o_l}, \sigma_1, \sigma_2, \sigma_3, \underline{e}, \overline{e}, \underline{\mu}, \overline{\mu}, \underline{\omega}, \overline{\omega}, \underline{\tau}, \overline{\tau}, \varphi, \rho\right)\right)$	
$s \leftarrow s_o$	// initialise current solution $-$ s as the initial solution $-$ s_o
$s^* \leftarrow s$	// initialise best solution $-$ s^* as the current solution
$H \leftarrow \{h(s)\}$	// initialise hash list
$T \leftarrow \overline{\omega} f(s^*) / \ln(1/\overline{\tau})$	// initialise the current temperature based on the cooling schedule
for $o_r \in \boldsymbol{o_r}$ **do**	// initialise removal operator weights to 1
$w_r \leftarrow 1$	
end for	
for $o_i \in \boldsymbol{o_i}$ **do**	// initialise insertion operator weights to 1
$w_i \leftarrow 1$	
end for	
$u \leftarrow 1$	// initialise segment index to 1
while $u \leq j$ **do**	// repeat for j segments
for $o_r \in \boldsymbol{o_r}$ **do**	
$p_r \leftarrow w_r / \sum_{r \in \Psi_r} w_r$	// update removal operator probability
end for	
for $o_i \in \boldsymbol{o_i}$ **do**	
$p_i \leftarrow w_i / \sum_{r \in \Psi_i} w_i$	// update insertion operator probability
end for	
for $o_r \in \boldsymbol{o_r}$ **do**	
$c_r \leftarrow 0$	// set removal operator count to 0
$\pi_r \leftarrow 0$	// set removal operator score to 0
end for	
for $o_i \in \boldsymbol{o_i}$ **do**	
$c_i \leftarrow 0$	// set insertion operator count to 0
$\pi_i \leftarrow 0$	// set insertion operator score to 0
end for	
$v \leftarrow 1$	// initialise iteration index to 1
while $v \leq n$ **do**	// repeat for n iterations
$o_r \xleftarrow{R_{p_r}} \boldsymbol{o_r}$	// randomly select a removal operator
$o_i \xleftarrow{R_{p_i}} \boldsymbol{o_i}$	// randomly select an insertion operator
$c_r \leftarrow c_r + 1$	// update removal operator count
$c_i \leftarrow c_i + 1$	// update insertion operator count
$\Lambda \sim U(0,1)$	
$\lambda \xleftarrow{R} \Lambda$	
$q \leftarrow \begin{bmatrix} (1-\lambda)\min\left(\underline{e}, \underline{\mu}\lVert s \rVert\right) \\ +\lambda \min(\overline{e}, \overline{\mu}\lVert s \rVert) \end{bmatrix}^{-}$	// set the size of removal and insertion operation
$s' \leftarrow o_i\left(o_r(q, s)\right)$	// remove and insert select number of customer nodes
if $f(s') < f(s^*)$ **then**	// if the new solution is better than the best solution then

$s^* \leftarrow s'$ // update the best solution to the current solution
$s \leftarrow s'$ // update the current solution to the new solution
$\pi_r \leftarrow \pi_r + \sigma_1$ // update removal operator score by σ_1
$\pi_i \leftarrow \pi_i + \sigma_1$ // update insertion operator score by σ_1
else if $f(s') < f(s)$ **then** // else if the new solution is better than the current solution then
$s \leftarrow s'$ // update the current solution to the new solution
if $h(s) \notin H$ **then** // if the solution does not exist in the hashed tabu list
$\pi_r \leftarrow \pi_r + \sigma_2$ // update removal operator score by σ_2
$\pi_i \leftarrow \pi_i + \sigma_2$ // update insertion operator score by σ_2
end if
else // else accept new solution with a small probability
$\Lambda \sim U(0,1)$
$\lambda \overset{R}{\leftarrow} \Lambda$
if $\lambda < \exp\left(\frac{f(s)-f(s')}{T}\right)$ **then**
$s \leftarrow s'$ // update the current solution to the new solution
if $h(s) \notin H$ **then** // if the solution does not exist in the hashed tabu list
$\pi_r \leftarrow \pi_r + \sigma_3$ // update removal operator score by σ_3
$\pi_i \leftarrow \pi_i + \sigma_3$ // update insertion operator score by σ_3
end if
end if
end if
$H \leftarrow H \cup \{h(s)\}$ // add the current solution to the hash list
$T \leftarrow \max\left(\varphi T, \underline{\omega} f(s^*)/\ln(1/\underline{\tau})\right)$ // update current temperature based on the cooling schedule
$v \leftarrow v + 1$ // update iteration index
end while
if $h \bmod k$ // after every k segments reset best current solution to best solution
$s \leftarrow s^*$
end if
for $o_r \in \boldsymbol{o_r}$ **do** // update removal operator weights
if $c_r \neq 0$ **then**
$w_r \leftarrow \rho\pi_r/c_r + (1-\rho)w_r$
end if
end for
for $o_i \in \boldsymbol{o_i}$ **do** // update insertion operator weights
if $c_i \neq 0$ **then**
$w_i \leftarrow \rho\pi_i/c_i + (1-\rho)w_i$
end if
end for
for $o_l \in \boldsymbol{o_l}$ // iteratively perform local search on the current solution
$s \leftarrow o_l(s, m)$
end for
if $f(s) < f(s^*)$ **then** // if the current solution is better than the best solution then
$s^* \leftarrow s$ // update the best solution to the current solution
end if
$H \leftarrow H \cup \{h(s)\}$ // add the current solution to the hash list
$u \leftarrow u + 1$ // update segment index
end while
return s^* // return the best solution