Optimizing Multi-Plant Capacitated Lotsizing and Scheduling Problem: A Hybrid Heuristic and Metaheuristic Approach

Ana Rita Mendes[a], Gonçalo Fonte[a], Gustavo Araújo[a] and Rodrigo Costa[a]

[a] *Faculdade de Engenharia da Universidade do Porto, Porto, Portugal*

ARTICLE INFO

Keywords:
Multi-Plant Capacitated Lot-Sizing and Scheduling Problem (MPCLSP)
Production Planning
Heuristics
Metaheuristics
Tabu Search
Iterated Local Search
Combinatorial Optimization

ABSTRACT

The Multi-Plant Capacitated Lotsizing and Scheduling Problem (MPCLSP) entails creating cost-effective production schedules across various plants and time periods while satisfying demand within capacity restrictions. Because of its combinatorial character, addressing this problem demands complex optimization techniques. This work provides a hybrid method that combines a constructive heuristic for rapidly generating feasible solutions, with Iterated Local Search (ILS) and Tabu Search metaheuristics for further optimization. The constructive heuristic promotes demand distribution across plants based on cost-effectiveness while controlling capacity and stock levels. The ILS and Tabu Search frameworks iteratively search neighborhoods of the solution space to reduce total costs while avoiding local optima. Computational experiments illustrate the method's effectiveness and robustness, resulting in substantial cost savings over baseline methods for numerous test scenarios. The findings demonstrate how the approach can be applied to complicated manufacturing scheduling circumstances.

1. Introduction

The MPCLSP (Multi-Plant Capacitated Lot-Sizing Problem) is a challenging combinatorial optimization problem that involves managing multiple plants, time periods, and products. Its main goal is to create an optimal production plan that minimizes total costs (including production, setup, transfer, and inventory costs), while ideally ensuring that demand for all items is met, on time, during each period.

MPCLSP has broad applications across industries. In the pharmaceutical industry, it optimizes drug manufacturing processes Jans and Degraeve (2007). Furthermore, in the chemical industry, it helps schedule petrochemical production at different fields, thus responding well to storage availability and transfer costs Shah (2005).

In the simplified version of the problem given, the products are treated independently, meaning that the production of one product does not influence the production of others, nor is there a required order for production. Due to the NP-hard nature of the problem, heuristic methods are necessary as the problem size becomes prohibitive for exact approaches. As a result, the primary aim was to construct and improve an heuristic, which would provide feasible solutions in a prompt manner and subsequently be improved through the iterative Tabu Search metaheuristic to further optimize these solutions.

During the development of the initial heuristic, however, it stood clear that meeting demand and capacity constraints would often result in infeasible problems. To address this, the capacity constraint was relaxed, enabling to analyze what adjustments would be necessary to meet the required demand. To account for this relaxation, the introduction of penalties for exceeding capacity was done, which increased the total costs incurred in the results.

The paper's structure begins with a theoretical contextualization of some concepts used throughout the study, in Section 2, followed by the problem characterization, mathematical formulations and a detailed explanation of the utilized methods in Section 3. The results achieved are presented in Section 4. Lastly, Section 5 provides a final discussion and concludes the paper.

2. Theoretical Contextualization

Optimization problems, especially in combinatorial domains, frequently necessitate advanced techniques to obtain high-quality solutions in reasonable computational time. These problems have severely huge solution spaces, which makes accurate techniques impractical in many circumstances. As a result, heuristic techniques have become essential tools for overcoming such obstacles. Valencia-Rivera, Benavides-Robles, Morales, Amaya, Cruz-Duarte, Ortiz-Bayliss and Avina-Cervantes (2024)

A constructive heuristic is a method for generating an initial feasible solution to an optimization problem, which can then be refined further using improvement heuristics. The basic idea is to create the solution incrementally, usually by adding components one at a time depending on a predetermined criterion, until a complete solution is achieved. Constructive heuristics should strike a compromise between simplicity and computing efficiency, using problem-specific knowledge to drive the solution design process. Although the answers they provide may not be optimum, they serve as a solid foundation for iterative improvement procedures.Oliveira and Carravilla (2009)

Metaheuristics are high-level problem-solving frameworks that guide and improve heuristic search processes while refining solutions given by constructive heuristics. One of the most effective frameworks for this is Iterated Local Search. This metaheuristic works by iteratively applying local search methods to investigate the neighborhood of solutions while systematically perturbing the search trajectory to avoid local optimums. Lourenço, Martin and Stützle (2009)

Tabu Search is a feature of the Iterated Local Search framework that improves the local search phase through the addition of memory structures. The Tabu list is a short-term memory system that prohibits the algorithm from revisiting recently examined solutions or steps, hence preventing cycles and stagnation and encouraging the reach of unexplored portions of the solution space. Another essential aspect of Tabu Search is the aspiration criteria, which allows the algorithm to override the Tabu list's limits if a solution fits specific circumstances, such as producing a better objective value than any previous solution discovered. This ensures that the search remains flexible and does not ignore potentially valuable solutions merely because they do not meet Tabu limitations.Glover (1990)

When paired with constructive heuristics, metaheuristics substantially enhance solution quality. While a constructive heuristic is a rapid and practical starting point, metaheuristics like Iterated Local Search and Tabu Search can repeatedly improve these solutions and lead the search to global optimality. This combination enables metaheuristics to use the capabilities of constructive heuristics while reducing their limitations, making them indispensable tools for handling difficult optimization issues.Johnson and McGeoch (1997)

3. Methods

The characterization of the problem began with an overview of the given instance, followed by a mathematical formulation that details the parameters, constraints, and the objective function required to achieve the goal of minimizing total costs.

First, an analysis of the provided costs was performed, which included operational costs from production (variable) and setup (fixed). These costs vary by product and plant. Production costs were defined as the expenses incurred for producing a single unit of a product in a specific plant. Setup costs, on the other hand, represented the fixed expense required to prepare a plant to produce any quantity of a specific product during any single period. Inventories and transfers between plants (producing items in a plant to fulfill demand from another one) were also permissible at a cost and differed for each product: inventory costs were given as the cost of keeping a unit in inventory from one period to the next, while transfer costs were given as the cost of transferring a unit between two plants. There were also established costs to transfer a unit between the same plant, which were assumed as costs of transportation between the plant and the delivery location.

Next, an analysis on the time-related data was conducted. This included operational times such as production and setup. The structure of the time data mirrored the one from the cost data, with values specified for each product and plant. The capacity was also given in total operating time available per plant in each period.

Lastly, the demand data to be met had three dimensions: period, product, and plant. It was provided as the number of units required for each product at each plant during specific periods.

This way, after thoroughly analyzing the given data, the mathematical formulation for the problem is as follows:

3.1. Parameters

I: Number of plants.
T: Number of periods.
P: Number of products.
setup_cost$_{p,i}$: Setup cost for product p at plant i.
inventory_cost$_{p}$: Inventory holding cost for a unit of product p.
prod_cost$_{p,i}$: Production cost of a unit of product p at plant i.
transfer_cost$_{i,j}$: Transfer cost of a unit between plants i and j.
d$_{p,i,t}$: Demand for product p at plant i in period t.
setup_times$_{p,i}$: Setup time for product p at plant i.
processing_times$_{p,i}$: Processing time for a unit of product p at plant i.
capacity$_{i,t}$: Available capacity at plant i in period t.

3.1.1. Indices

i represents the production plant index.
j represents the demand plant index.
t represents the production period.
u represents the demand period.
p represents the product index.

3.2. Constructive Heuristic

Below, it is provided a thorough explanation on how our constructive heuristic works.

3.2.1. Candidate List Generation

In this primary analysis, the goal was to determine the most cost-effective production plant for each demand block across various products, plants, and periods. The process involved multiple steps: calculating average demand, defining a cost function, ranking production plants by cost and displaying the rankings for each demand block. Below is the detailed explanation of how the rankings were generated.

First, the average demand for each product across all plants and periods was calculated. This was essential for the subsequent cost calculations. The average demand per product, $\bar{d}_p$, for each product p is given by:

$$\bar{d}_p = \frac{1}{I \cdot P} \sum_{i=1}^{I} \sum_{j=1}^{P} d_{p,i,t} \tag{1}$$

where:

- I is the number of plants,
- P is the number of periods,

- $d_{p,i,t}$ is the demand for product p at plant i in period t.

After this, the cost to produce a given demand block (defined by a product, demand plant, and demand period) in each production plant was computed, and all plants were ranked according to their cost.

$$\begin{aligned} C(p,i,t,j) = &\ \text{prod\_cost}(p,j)\cdot\bar{d}_p \\ &+ \text{setup\_cost}(p,j) \\ &+ \text{transfer\_cost}(j,i)\cdot\bar{d}_p \end{aligned}$$

For each product p, demand plant i, and period t, the production plants were sorted in ascending order of cost (cheapest production plant is ranked first), forming the ranking matrix, $\mathbf{ranking}(p,i,t)$, which holds the indices of production plants sorted by their respective costs.

$$\mathbf{ranking}(p,i,t) = \text{argsort}(C(p,j,t,i)) \quad \text{for } i = 1,2,\ldots,I$$

Finally, the rankings were displayed in a user-friendly format, showing the production plant rankings and the corresponding costs for each demand block. The formatted output offers a first view into the most cost-effective production plants to meet the demand at each plant and period, prior to any allocation (at this stage, inventory is still not taken into account and setups are always considered).

3.2.2. Demand Allocation

The allocation of products to production plants and periods was carried out using a backward loop, starting from the last period and ending at the first. This approach ensured the optimization of production capacity and minimization of costs, making sure that no demand was left to be met at the end of the last period. The main steps of the allocation algorithm are as follows:

1. **Initialization:**
    - The allocation matrix was initialized as a zeros matrix, with shape (P,I,I,T,T), given the fact that it needed to store information about the quantity allocated of all demand blocks, plus the period and plant where they were produced to account for inventory and transfers, respectively.
    - The remaining capacity for each plant and period was initialized as a copy of $\mathbf{capacity}_{i,t}$.
    - Inventory levels and cost variables (setup costs, transfer costs, production costs and inventory costs) were initialized as zero.
2. **Cost Calculation:** At this phase, the total cost function of delivering the demand block was calculated by:

$$\begin{aligned} C(p,i,t,j,u) = &\ \text{prod\_cost}(p,j)\cdot\bar{d}_p \\ &+ \text{setup\_cost}(p,j) \\ &+ \text{transfer\_cost}(j,i)\cdot\bar{d}_p \\ &+ \text{inventory\_cost}(p) \\ &\cdot \text{inventory}(p,j,t) \end{aligned}$$

   This formula differs from the one of the last subsection in the way that setup costs are now applied only if there was no prior production allocation for the plant during the specific period, that means, if a quantity of some product was already allocated to be produced at that plant, the setup cost is counted as zero additional cost. Furthermore, the costs were adjusted with a penalty term if the remaining capacity of a plant became insufficient.
3. **Demand Satisfaction:** The algorithm iteratively assigns production to the highest-ranked plants, while ensuring that the remaining capacity of the plant is sufficient for the pretended quantity to produce.
   However, if the best-ranked plant isn't able to satisfy the demand, the model moves to the next plant on the ranking or allocates the demand to previous periods, if feasible. This feasibility was estimated based on the expected demand that the plants from the previous period would have to satisfy.
4. **Handling Excess Demand:** If a plant's capacity is insufficient to meet all the demand, the algorithm selects the plant with the least penalty for exceeding capacity, in the period in which the demand existed. The remaining demand is allocated in that plant, and a penalty cost is added.
5. **Updates:** After each allocation, the remaining capacity, inventory levels, and costs are updated. Any unfulfilled demand is carried forward to the next iteration (previous period).
6. **Output:** The allocation matrix is updated with the assigned quantities for each product, production plant, demand plant, and period. The total costs are also computed and logged.

This procedure ensured that production was allocated efficiently while adhering to capacity constraints, minimizing costs, and handling excess demand when necessary. This way, it was possible to achieve a constructive, greedy deterministic solution.

3.2.3. Pseudo-code

The following pseudo-code provides a compact representation of how the constructive heuristic works, as explained in great detail up until this point.

1. **Initialization:**
    - allocation = 0 (5D matrix for production allocation)
    - remaining_capacity = capacity (plant capacities)
    - inventory = 0 (inventory tracking)
    - penalty_per_unit = 3 * production_costs / processing_times
    - setup_cost, production_cost, transfer_cost, inventory_cost = 0
    - f_demand = sum of all demand
2. **Main Loop: Reverse Period Allocation**
   FOR period **IN** reverse order (last to first):
   FOR each product:

```
    • FOR each demand_plant:
      (a) Set remaining_demand = demand[product][demand_plant][period]
      (b) WHILE remaining_demand > 0:
          – Calculate costs for all production plants: cost = production_cost + penalty (if no capacity)
          – Rank plants by cost (ascending).
      (c) FOR production_plant IN ranked plants:
          – IF remaining_capacity allows production:
              ∗ Calculate max_possible = remaining_capacity / processing_times
              ∗ Set production_quantity = min(remaining_demand, max_possible)
          – IF production_quantity > 0:
              ∗ Deduct setup cost if first production in plant.
              ∗ Update allocation, remaining_capacity, and remaining_demand.
              ∗ Update production_cost, setup_cost, and transfer_cost.
          – Print allocation details.
          – IF remaining_demand == 0: BREAK
      (d) IF no plants can satisfy demand in the current period:
          – Check plants with remaining capacity:
              ∗ Allocate as much as possible.
          – IF still unsatisfied and periods remain:
              ∗ Backward allocate to earlier periods.
              ∗ Include setup and inventory costs.
          – IF still unmet demand in period 0:
              ∗ Allocate at the cheapest plant with penalty.
3. Handle Residual Demand
  (a) FOR remaining unmet demand:
      • Allocate to plants with the least capacity violations.
4. Final Output
  (a) Print remaining unsatisfied demand (f_demand).
  (b) Print all allocations and cost summaries.
```

3.3. Improvement Heuristic

After obtaining a good quality initial solution, and feasible within the constraints that were relaxed, the next step was the improvement heuristic. At this stage, the objective was to explore the neighborhood of the initial solution, and iteratively try to keep minimizing the total cost of the allocation plan.

The constructive heuristic's output was a five-dimensional matrix. However, working with this sort of input for the next step would become computationally expensive due to the high memory requirements in each iteration.

To overcome this, a simplified binary matrix was derived from the allocation matrix, constructed with shape (P, I, I, T), and containing the value of 1 if a specific product p is being produced at the plant i and demanded by the plant j in period t, and 0 otherwise. The period in which demand needed to be fulfilled didn't need to be explicitly included in the binary matrix, since the quantities required to meet demand in that period would be determined later by the new heuristic, based on the available production periods with value 1. By abstracting away the period when the production is needed, and focusing only on whether production occurs in that period, the matrix reduces complexity without losing essential information for exploring feasible neighborhoods.

Consequently, neighbors could now be easily generated by flipping the value of a single cell in the binary matrix, changing the production status of a product at a plant in a given period (Hamming Distance = 1).

Therefore, what was left to determine in the neighbor solutions were the quantities of each product allocated to each plant in each period, in order to find the combination that led to the lowest possible cost. This final step was done through a linear problem, in which the different binary matrices generated were the input, and works as follows:

3.3.1. *Decision Variables*

$\mathbf{x}_{p,i,j,t}$: Quantity of product p produced at plant i demanded from plant j in period t.
capacity_exc$_{i,t}$: Capacity exceedance of plant i in period t.
inventory$_{p,i,t}$: Inventory level of product p at plant i in period t.

3.3.2. *Objective Function*

$$\begin{aligned}\min \quad & \sum_{p,i,t} \text{setup\_costs}_{p,i} \cdot \text{setup}_{p,i,t} \\ & + \sum_{p,i,j,t} \text{production\_costs}_{p,i} \cdot x_{p,i,j,t} \\ & + \sum_{p,i,j,t} \text{transfer\_costs}_{i,j} \cdot x_{p,i,j,t} \\ & + \sum_{p,i,t} \text{inventory\_costs}_{p} \cdot \text{inventory}_{p,i,t} \\ & + \sum_{p,i,t} \text{capacity\_exc}_{i,t} \cdot \text{capacity\_penalty}_{p,i}\end{aligned} \tag{2}$$

3.3.3. *Constraints*

- **Demand Satisfaction:**

$$\sum_i x_{p,i,j,t} + \text{inventory}_{p,j,t-1} \geq \text{demand}_{p,j,t}, \quad \text{for } t > 0 \tag{3}$$

$$\sum_i x_{p,i,j,t} \geq \text{demand}_{p,j,t}, \quad \text{for } t = 0 \tag{4}$$

- **Inventory Balance:**

$$\text{inventory}_{p,j,t} \geq \sum_i x_{p,i,j,t} + \text{inventory}_{p,j,t-1} - \text{demand}_{p,j,t}, \quad \text{for } t > 0 \tag{5}$$

$$\text{inventory}_{p,j,t} = 0, \quad \text{for } t = T \tag{6}$$

$$\text{inventory}_{p,j,t} \geq \sum_i x_{p,i,j,t} - \text{demand}_{p,j,t}, \quad \text{for } t = 0 \tag{7}$$

- **Production Setup:**

$$\sum_j x_{p,i,j,t} = 0, \quad \text{if setup}_{p,i,t} = 0 \tag{8}$$

$$\sum_j x_{p,i,j,t} \geq 1, \quad \text{if setup}_{p,i,t} = 1 \tag{9}$$

- **Capacity Constraints:**

$$\begin{aligned}&\text{capacity\_exc}_{i,t} \geq \sum_{p,j} x_{p,i,j,t} \cdot \\ &\text{processing\_times}_{p,i} + \sum_p \text{setup}_{p,i,t} \cdot \\ &\text{setup\_times}_{p,i} - \text{capacity}_{i,t}\end{aligned} \tag{10}$$

- **Total Demand Match:**

$$\sum_{i,j,t} x_{p,i,j,t} = \sum_{j,t} \text{demand}_{p,j,t}, \quad \forall p \tag{11}$$

Note: the setup matrix referenced in the formulation above is the result of a function that transforms the binary matrix into an even more simplified one of three dimensions, shaped $((P,I,T))$, considering only machine setups in each production period. This way, $setup_{p,i,t}$ is not a decision variable in the formulation, since it comes directly from the binary matrix, which is the one argument for the linear programming function.

The objective function 2 was, once again, to minimize the total cost.

Furthermore, constraints 3 and 4 ensure total demand satisfaction: the sum of all production of product p demanded from plant j in period t, plus the inventory of p coming from the previous period and needed for j (when period t is not the first one), must be higher or equal to the demand for p at j in t. Constraints 5, 6 and 7 refer to inventory balance: a period's inventory should be equal to the total production of that period t plus the inventory coming from the last period, minus the total demand to be met in t. Exceptions are the first period, where there is still no inventory coming from previous periods, and the last one, where no final inventory should remain. Lastly, constraints 8 and 9 impose that production can only happen if a setup is performed, and constraint 10 ensures that capacity exceedance happens when the total used time for production and setups in a period t and plant i is superior to that of the theoretical capacity.

3.4. Tabu Search Implementation

After all this, the aim was to further optimize the allocation plan iteratively by exploring neighborhoods of the current best solution, and, temporarily, allowing worse moves to escape local optima, so the Tabu Search metaheuristic was implemented through a loop with a stopping condition based on no improvement.

1. **Initialization:**
   - Start with the setup matrix obtained from the constructive heuristic and solve the linear problem (LP) to determine the current solution's cost and quantities allocated $(x_{p,i,j,t})$.
   - Initialize the Tabu matrix with zeros, meaning no initial move restrictions, and dimensions $((P,I,T))$. This matrix will track the *tenure* of recent moves: a value greater than 0 in a specific position will indicate that the corresponding move (flip) in the binary matrix is temporarily prohibited for that value number of iterations.
2. **Beginning of Loop - Neighborhood Generation:**
   - At each iteration, a neighbor solution is generated by randomly flipping a position in the setup matrix, as explained before.
   - The flip position is identified by comparing the new neighbor's initial setup matrix with the current one.
3. **Evaluation of Neighbor Solutions:**
   - The cost of each generated binary solution is calculated by solving the LP.
   - If the move that led to the specific solution is a Tabu, the solution isn't allowed, unless it led to a better solution than the current best known total cost (aspiration criteria).
4. **Tabu List Updates:**
   - If a move is accepted (non-Tabu, or meets aspiration criteria), the Tabu tenure (e.g. 10 iterations) for the flip position in the Tabu list is updated to prevent its immediate reversal.
   - The tenure of every position in the Tabu matrix is decremented at each iteration.
5. **End of Loop - Stopping Conditions:**
   - The loop terminates when a predefined number of iterations without improvement is reached.
   - If the current best known solution hasn't improved at the end, the "best worst" (lowest cost solution found in the cycle) solution is selected as the final output.

The constructed loop was then executed for cycles of varying numbers of iterations (as detailed in Section 4), and the solutions from each loop output (both new best solutions and "best worsts") were stored for result analysis. Additionally, counters were created throughout the cycle to keep track of metrics such as the frequency in which the solution was worsened to escape local optima and the number of times aspirational criteria were met.

Finally, the cycle allowed to keep, most importantly, the final best encountered solution, which achieved the minimum cost obtained during the attempt.

4. Results

Here, a performance evaluation of the proposed approach for solving the Multi-Plant Capacitated Lotsizing Scheduling Problem was conducted, as well as its ability to handle different instances effectively. This includes assessing the quality of the produced solutions and the time required to get to a good solution.

A typical instance for the MPCLSP was used for testing, featuring multiple periods, varying demands, costs and capacities, which is what should be expected in the real-world production planning challenges. In this test instance, 15 products were to be produced at 4 production plants in 6 different time periods.

The computational experiments for this study were conducted on a Hewlett-Packard laptop equipped with a 12th Gen Intel(R) Core (TM) i7-1260P processor operating at 2.10 GHz. The system features 16.0 GB of installed RAM and operates on a 64-bit Windows operating system. Python was the chosen programming language to implement and execute the code.

The application of the constructive heuristic to the previously presented dataset resulted in an overall total cost of 113,929.56 monetary units (MU), from which costs of setup accounted for 19,421.00. The amount designated to producing the required units totaled 77,220.73 and, in the activity of moving the products from the production plant to the site where they are demanded, 11,312.41 MU were spent. On top of that, a penalty cost of 5,975.42 was incurred due to exceeding the production capacity of the plants. These results establish a baseline solution, which, while feasible, leaves room for improvement, particularly in reducing penalty.

From that same starting point, the model was ran multiple times, varying the number of iterations in some of them, and also assessing the results generated from different Tabu tenures throughout all the runs. The best result achieved (corresponding to the lowest cost) across all runs was 97258.47 MU by doing 150 iterations and accepting a worse solution when no improvement was achievable for the first 100 neighbors generated, with a Tabu tenure of ten iterations.

On table 1, the results of average gap and run time were given for different synthetic instances. The optimality gap is given by the following formula:

$$\text{Gap}(\%) = \frac{|\text{Best Known Solution} - \text{Lower Bound}|}{|\text{Best Known Solution}|} \times 100$$

The dimension of the instances is given by: number of plants x number of periods x number of products.

Graphical representations were also made in order to visually analyze the evolution of the cost, and the gap of each model. These are included in the Appendix sections A and B.

An analysis was performed regarding the impact of increasing the number of iterations when searching for the next neighbor in the solution space. These results are presented in table 2, and were estimated only for the original instance given.

Table 1
Performance Metrics: Average Gap and Run Time.

Dimension	Avg. Gap	Run Time (s)
4 x 6 x 15 - 20	0.2391	1423
4 x 6 x 15 - 10	0.2425	1308
3 x 5 x 13 - 20	0.2024	1032
3 x 5 x 13 - 10	0.2004	1239
5 x 5 x 9 - 20	0.02437	329
5 x 5 x 9 - 10	0.02188	307

Table 2
Performance Metrics: Optimal Cost, Average Gap, and Run Time for Varying Max Iterations.

Max Iterations	Opt. Cost	Avg. Gap	Run Time (s)
10	102909.87	0.2621	302
25	102059.72	0.2537	512
50	99192.79	0.2409	837
75	99621.60	0.2419	1337
100	97708.66	0.2391	1423

On section C of the Appendix, included graphs showcase the behavior of the best cost found on all of these models.

5. Discussion

Considering the obtained results, some conclusions can be made on what regards the influence of the instance dimension, maximum number of iterations, best costs obtained, and run time.

Firstly, it stood clear that increasing the number of iterations when exploring the neighborhood leads to higher run times. This can be explained since, at some point, it becomes harder to find neighbors that lead to better costs, and the model takes longer times in the searching loop, trying to find these more cost-efficient solutions. Additionally, it was also possible to understand that models with larger number of iterations also led to better costs overall.

Furthermore, when working with different instances, the results of average gap and run time seemed to differ a lot. For example, the 5x5x9 instance led to a way better solution in a quicker time. This can be explained through the fact that this instance had way more capacity available than needed, when compared with others. This made the model seem both more effective and efficient, finding a great-quality solution. Generally, it could be said that for higher values of the ratio given by $\frac{\text{Total Capacity}}{\text{Total Demand}}$ the model is faster in finding a solution closer to the optimal.

Lastly, concerning the Tabu list, for the two values used for its length (10 and 20), neither of them seemed to lead to better solutions of average gap and run time. When discussing what might lead to this situation, it was determined that, probably, the delta between the two values wasn't big enough, or the randomness of the neighbor search doesn't allow for a strict position in this topic. The recommendation sits to take this as a hyperparameter, and test it in each specific working instance.

References

Glover, F., 1990. Tabu Search: A Tutorial. The Institute of Management Sciencies.

Jans, R., Degraeve, Z., 2007. Modeling industrial lot sizing problems: A review. International Journal of Production Research .

Johnson, D.S., McGeoch, L.A., 1997. The traveling salesman problem: A case study in local optimization, in: Aarts, E.H.L., Lenstra, J.K. (Eds.), Local Search in Combinatorial Optimization. John Wiley & Sons, New York, pp. 215–310.

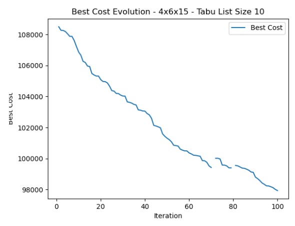
Lourenço, H.R., Martin, O.C., Stützle, T., 2009. Iterated local search. Unpublished Universitat Pompeu Fabra, Barcelona, Spain; Université Paris-Sud, Orsay, France; Darmstadt University of Technology, Darmstadt, Germany. Email: helena.ramalhinho@econ.upf.es, martino@ipno.in2p3.fr, stuetzle@informatik.tu-darmstadt.de.

Oliveira, J.F., Carravilla, M.A., 2009. Heuristics and Local Search. Version 2 ed., FEUP. C©2009, 2001.

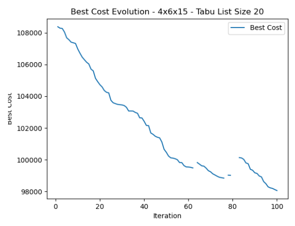
Shah, N., 2005. Process industry supply chains: Advances and challenges.

Valencia-Rivera, G.H., Benavides-Robles, M.T., Morales, A.V., Amaya, I., Cruz-Duarte, J.M., Ortiz-Bayliss, J.C., Avina-Cervantes, J.G., 2024. A systematic review of metaheuristic algorithms in electric power systems optimization. Applied Soft Computing 150, 111047. URL: https://doi.org/10.1016/j.asoc.2023.111047, doi:10.1016/j.asoc.2023.111047.

A. Graphical Analysis of the Cost Evolution

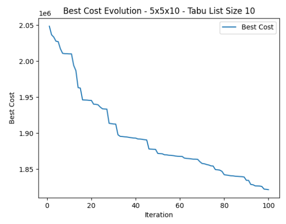


(a) Tabu Tenure 10

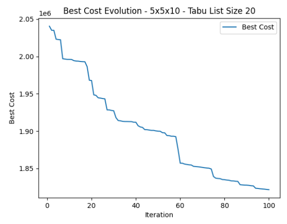


(b) Tabu Tenure 20

Figure 1: Comparison of Cost Evolution for Different Tabu Tenures - Original Instance.

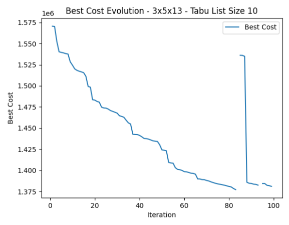


(a) Tabu Tenure 10

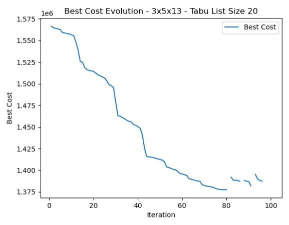


(b) Tabu Tenure 20

Figure 2: Comparison of Cost Evolution for Different Tabu Tenures - Instance 1.



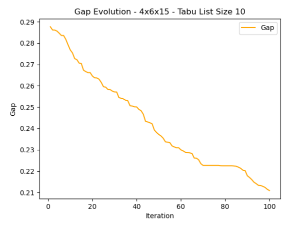
(a) Tabu Tenure 10



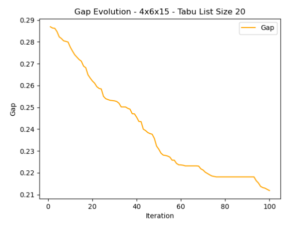
(b) Tabu Tenure 20

Figure 3: Comparison of Cost Evolution for Different Tabu Tenures - Instance 3.

B. Graphical Analysis of the Gap Evolution

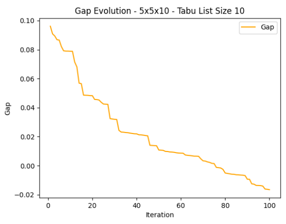


(a) Tabu Tenure 10

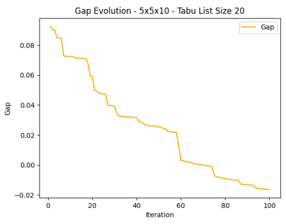


(b) Tabu Tenure 20

Figure 4: Comparison of Gap Evolution for Different Tabu Tenures - Original Instance.

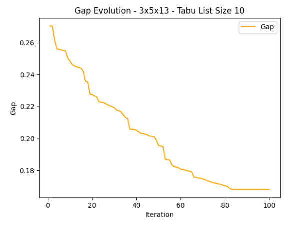


(a) Tabu Tenure 10

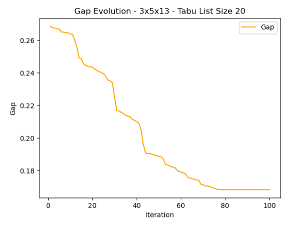


(b) Tabu Tenure 20

Figure 5: Comparison of Gap Evolution for Different Tabu Tenures - Instance 1.



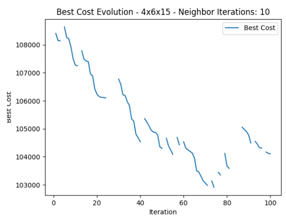
(a) Tabu Tenure 10



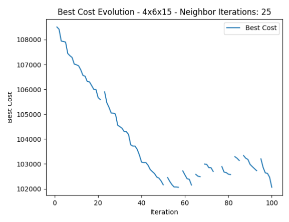
(b) Tabu Tenure 20

Figure 6: Comparison of Gap Evolution for Different Tabu Tenures - Instance 3.

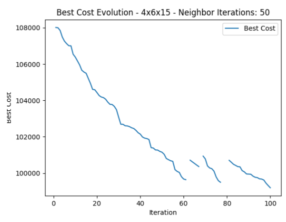
C. Cost Evolution on Different Maximum Iterations Values for Neighbor Search



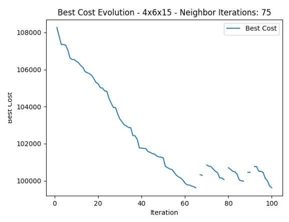
(a) Max. Iterations: 10



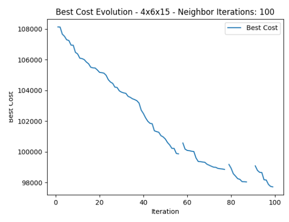
(b) Max. Iterations: 25



(c) Max. Iterations: 50



(d) Max. Iterations: 75



(e) Max. Iterations: 100

Figure 7: Comparison of Cost Evolution for Different Maximum Iterations in Neighbor Search.