Study about a Multi-start metaheuristic approach for the SALB3PM

Thiago G. Araujo, Matthieu Py, Laurent Deroussi and Nathalie Grangeon

***Abstract*— With growing emphasis on sustainable manufacturing, we address the energy-aware *Simple Assembly Line Balancing Problem with Power Peak Minimization* (SALB3PM). We propose a metaheuristic to solve the SALB3PM. Our metaheuristic combines (1) a multi-start framework with three neighborhood operators (insertion, swap, delay incrementing) and (2) a dynamic penalty mechanism for handling cycle-time infeasibility. We designed and compared four algorithm variants to evaluate the impact of components. We conducted computational experiments on benchmark instances. The results show that our best variant achieves an average optimality gap of 2.38% when considering known optima, and optimal solutions were achieved in all runs for over 50% of instances. The approach contributes to sustainable manufacturing through energy-efficient production line design.**

I. INTRODUCTION

Assembly lines represent a fundamental production system characterized by a linear arrangement of workstations. In this configuration, products move sequentially between stations, with specific tasks executed at each workstation until assembly completion. These systems are typically modeled as Simple Assembly Line Balancing Problems (SALBP) [1], which have been extensively studied in the literature [2], [3], [4].

The urgent need to address climate change has made reducing CO_2 emissions a critical priority. Recent data from the International Energy Agency (IEA) reveals that the industrial sector accounts for 45% of global CO_2 emissions and 39% of total energy consumption [5]. This environmental challenge has motivated the development of energy-aware SALBP variants that incorporate power consumption considerations.

A notable example is the SALBP with Power Peak Minimization (SALB3PM) introduced by [6]. In this variant:

- Each task has constant energy consumption independent of its workstation assignment
- The objective is to minimize power peak energy consumption across all time units

Exact methods have been proposed for SALB3PM:

- An Integer Linear Programming (ILP) formulation [6]
- A SAT-based approach using iterative solver calls [7]
- A MaxSAT formulation with alternative boolean representations [8]

While SAT and MaxSAT methods demonstrate superior performance compared to ILP formulations, their effectiveness diminishes as the problem scale increases, and they often fail to reach optimal solutions.

Given these limitations, researchers have explored metaheuristic approaches for a simplified variant, SALB3PM with Earliest Starting Dates (SALB3PM-ESD), which only works with semi-active schedules:

- Multi-start Evolutionary Local Search (MS-ELS) [9]
- ILP-based decoder for permutation-based heuristics applied to a simple Simulated Annealing algorithm [10]

Previous metaheuristic works have focused on the SALB3PM-ESD, therefore addressing semi-active schedules [9], [10]. Unfortunately, the set of semi-active schedules does not necessarily contain an optimal solution for SALB3PM. To address this gap, we propose a metaheuristic for the SALB3PM that incorporates task delays through the following components:

- A multi-start framework with local search
- A three vector-based representation
- Three neighborhood moves:
  - Task insertion
  - Tasks swap
  - Delay increment
- Priority-based solution initialization with semi-random prioritization
- Dynamic penalty for cycle-time violations, randomly adjusted at each iteration.

The remainder of this paper is organized as follows. Section II formally defines SALB3PM. Section III details our metaheuristic approach. Section IV presents computational results, and Section V discusses conclusions and future work.

II. PROBLEM DESCRIPTION

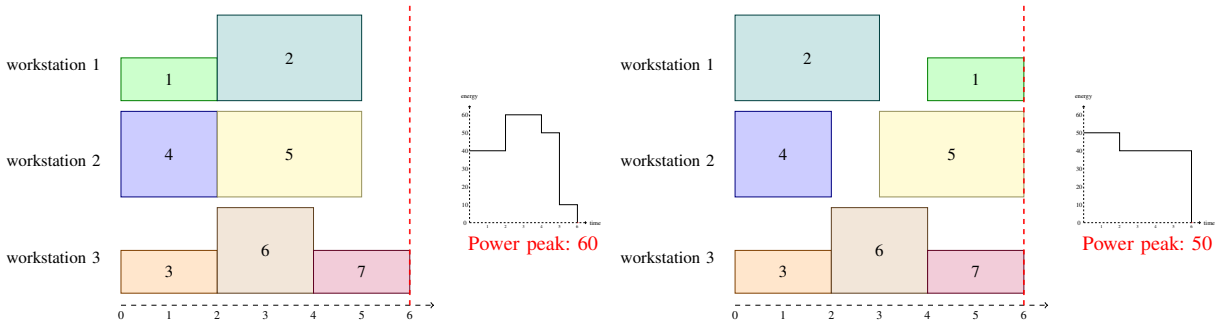
An assembly line consists of a set $\mathcal{M} = \{1, \ldots, m\}$ of workstations arranged in linear sequence. The manufacturing process requires executing a set $\mathcal{O} = \{1, \ldots, n\}$ of indivisible operations, referred to as tasks.

Due to inherent production constraints, certain tasks have precedence requirements that must be satisfied before execution. These precedence constraints form a partial ordering of tasks, which could be represented as a directed acyclic graph $G = (\mathcal{O}, E)$, where:

- Nodes $i \in \mathcal{O}$ correspond to tasks
- Edges $(i, j) \in E$ indicate that task i must complete before task j begins (denoted $i \prec j$)

Formally, for any precedence relation $i \prec j$, either (1) task i must be assigned to a workstation located before on the production line, or (2) to the same workstation as task j. This implies that the tasks must be partitioned into disjoint

Thiago G. Araujo, Matthieu Py, Laurent Deroussi, and Nathalie Grangeon with Université Clermont Auvergne, CNRS, Clermont Auvergne INP, Mines Saint-Étienne, LIMOS, 63000 Clermont-Ferrand, France {thiago.giachetto_de_araujo, matthieu.py, laurent.deroussi, nathalie.grangeon}@uca.fr



(a) Feasible solution (b) Optimal solution

Fig. 1: Examples of solution for the SALB3PM

sets $\{S_k\}_{k\in\mathcal{M}}$ such that for any $i \in S_a$ and $j \in S_b$ with $i \prec j$, we have $a \leq b$.

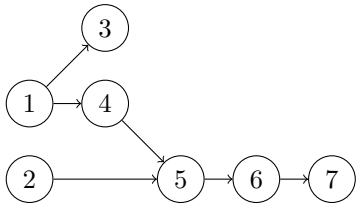


Fig. 2: Precedence graph

Figure 2 illustrates a precedence graph with seven tasks. For instance, task 6 requires the completion of its direct (task 5) and indirect predecessors (tasks 1, 2, and 4). Task 7 could begin after task 6 ends.

The Simple Assembly Line Balancing Problem (SALBP) consists of assigning each task to exactly one workstation while satisfying all precedence constraints. Each task $j \in \mathcal{O}$ has an associated processing time t_j (independent of workstation assignment). The total processing time of workstation k is given by:

$$t(S_k) = \sum_{j\in S_k} t_j \tag{1}$$

The amount of time workstations can spend executing tasks is called cycle time. Given a fixed cycle time c, a solution obeys the cycle time constraints if:

$$t(S_k) \leq c \quad \forall k \in \mathcal{M} \tag{2}$$

Different SALBP variants exist based on the chosen objective function:

- **SALBP-1**: Given c, minimize the number of workstations m
- **SALBP-2**: Given m, minimize the cycle time c
- **SALBP-F**: Given m and c, determine if a feasible line balance exists or not
- **SALBP-E**: Maximize efficiency by minimizing $m \times c$ [3]

The SALBP with Power Peak Minimization (SALB3PM) is a variant of SALBP-F with additional energy constraints, where:

- The number of workstations m and cycle time c are given parameters
- Each task $j \in \mathcal{O}$ has a fixed power consumption w_j (independent of workstation assignment)
- Instantaneous power consumption at time t is the sum of the power consumption of all tasks executed at time t
- The objective is to find a feasible task assignment and schedule that minimizes peak power consumption
- It is necessary to schedule tasks, being necessary to reinforce the precedence definition. If task i precedes task j and both are assigned to the same workstation, then task i must start before j.

TABLE I: Processing time and power consumption of tasks

Task	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Processing time	2	3	2	2	3	2	2
Power consumption	10	20	10	20	20	20	10

Example 1: We consider a SALB3PM instance with three workstations and a cycle time of 6. Table I presents the processing time and power consumption of each task. The precedence constraints are the same as Figure 2. Figures 1a and 1b present examples of solutions. Each rectangle represents a task where power consumption is symbolized by height and the processing time by width.

Figure 1a illustrates a feasible solution. At time $t = 0$, tasks 1, 4, and 3 begin execution, resulting in an initial power consumption of 40 units. When $t = 2$, tasks 1, 4, and 3 complete while tasks 2, 5, and 6 start, increasing power consumption to 60 units. At $t = 4$, task 6 finishes, and task 7 begins, reducing power consumption to 50 units. Finally, at $t = 5$, tasks 2 and 5 terminate, leaving only task 7 active with a power consumption of 10 units until the cycle completes.

An optimal solution is shown in Figure 1b. It can be observed that:

- The three tasks with the highest power consumption are not all executed at the same time
- The difference between the maximum and the minimum total power consumption is small compared to the feasible solution in Figure 1a
- Incorporating delays can reduce the power peak consumption.

III. A Multi-Start Metaheuristic for SALB3PM

As stated before, the SALB3PM is a variant of an NP-hard problem, the SALBP-F [1]. Therefore, finding optimal solutions for the SALB3PM is difficult, which motivates our investigation of a metaheuristic approach [6], [7]. The proposed metaheuristic is based on a multi-start framework that relies on semi-random solution generation, a local search using three different neighborhood moves, and a dynamic penalty mechanism.

A. Multi-Start Algorithm

Originally, multi-start procedures were used to exploit a local search procedure. Multiple initial solutions are randomly generated, and the local search procedure is applied [11]. Algorithm 1, adapted from [11], shows the principal components of a multi-start procedure for minimization problems. The algorithm requires two inputs: an evaluation function $f(\cdot)$ and a stopping criterion. At line 2, the best solution S is initialized. At line 4, a random value is sampled from a uniform distribution for use in the evaluation function (detailed in Section III-E). At lines 5–6, a semi-random solution is generated, and a local search procedure is applied. At lines 7–9, S is updated if the current solution is better. The main loop (3–10) executes until satisfying the stopping criteria. Finally, the best solution S is returned.

Algorithm 1 Multi-start algorithm

```
1:  function SEARCH(f(·), number_runs)
2:      S ← ∅, f(S) ← ∞
3:      while stopping criterion not satisfied do
4:          W ← U(0, max_{j∈O}(w_j))
5:          actual_sol ← randomized_solution()
6:          actual_sol ← local_search(actual_sol)
7:          if f(actual_sol) < f(S) then
8:              S ← actual_solution
9:          end if
10:     end while
11:     return S
12: end function
```

B. Solution representation

Solution representation is a key factor in successfully applying metaheuristics to optimization problems. Building upon the representation used in [9], which employed an assignment vector and an operation sequence vector, we extend this approach by introducing a third vector, D, which is our contribution. Our complete solution representation consists of three distinct vectors:

- **Assignment vector**: $A = (a_1, a_2, \ldots, a_n)$, where $a_i \in \mathcal{M}$ denotes the workstation assigned to task i
- **Tasks order**: $V = (v_1, v_2, \ldots, v_n$ specifies the processing order, where v_i represents the task at sequence position i
- **Delay vector**: $D = (d_1, d_2, \ldots, d_n)$, where $d_i \geq 0$ indicates the idle time between the completion of the task that precedes task i and its own start time on the same workstation.

Index	1	2	3	4	5	6	7
Assignment (A)	1	1	3	2	2	3	3
Order (V)	2	1	4	5	3	6	7
Delay (D)	1	0	0	0	1	0	0

Fig. 3: Representation of the solution of Figure 1b

During decoding, the solution is reconstructed through three phases: (1) workstation assignment via A, (2) task ordering within workstations through V, and (3) D determines the inter-task delays. After this decoding procedure, it is possible to determine when a task starts and ends.

C. Generation of solutions

Constructive heuristics with priority rules for the SALBP have been extensively studied in the literature [2], [4]. These priority-based methods assign weights to tasks to rank them. Since our multi-start algorithm requires diverse, good-quality initial solutions, we use a semi-randomized version of the longest processing time heuristic for solution generation [4].

$$\text{priority}(j) = (1 - \alpha)x + \alpha \frac{t_j - t_{min}}{t_{max} - t_{min}} \tag{3}$$

Equation 3 defines the priority rule for each task, where:

- $x \in [0, 1]$ is a uniformly distributed random value generated per task
- $\alpha \in [0, 1]$ controls the randomness degree. If $\alpha = 0$, the priorities are completely random. On the opposite way, for $\alpha = 1$, the priorities are greedy (and deterministic).
- t_{min} and t_{max} represent the minimum and maximum processing times, respectively

Each generated solution satisfies the following conditions: (1) precedence constraints are respected, (2) delay of all tasks is initialized to zero, and (3) cycle time constraints are enforced for all workstations except the last one. Some definitions are necessary to understand the assignment procedure. A task is considered *available* when all its direct predecessors have been assigned. Task j can be assigned to workstation k if:

- All predecessors are assigned to workstations $1, \ldots, k$, and
- The workstation's total processing time plus t_j does not exceed the cycle time (except for the last workstation)

The assignment procedure (Algorithm 2) utilizes the following components:

- $\mathcal{T}$: Set of unassigned tasks
- k: Current workstation reference
- CT: Available time
- LCT: Set of available tasks

The assignment function operates as follows. The algorithm receives as input a function $priority\_rule(\cdot)$ that returns the priority of a given task. Initialization occurs at Line 2, where $\mathcal{T}$ receives all tasks, the workstation reference k is set to 1, the available time CT is initialized to the cycle time c, and the assignment vector A is initialized with n zeros. The main loop (lines 3–16) iterates until all tasks are assigned ($\mathcal{T} = \emptyset$). Line 4 populated LCT with all assignable tasks.

When LCT contains tasks (Lines 6–9), the procedure: (1) selects the highest-priority task ($next$), (2) removes if from $\mathcal{T}$, (3) assigns it to workstation k, and (4) decrements the available time CT by t_{next}. Line 11 resets CT to c when no tasks are assignable to the current workstation ($LCT = \emptyset$). The workstation reference k increments at Line 13 when $k < m$, maintaining the number of workstations limited to m. Any remaining unassigned tasks are assigned to the last workstation ($k = m$), even if this violates the cycle time constraint.

Algorithm 2 Assignment procedure

Require: Cycle time (c), precedence relationships, tasks' processing time, number of workstations (m), and number of tasks (n)

1: **function** TASKS_ASSIGNMENT($priority\_rule(\cdot)$)
2: $\mathcal{T} \leftarrow \{1, \ldots, n\},\ k \leftarrow 1,\ CT \leftarrow c,\ A \leftarrow (0, \ldots, 0)$
3: **while** $\mathcal{T} \neq \emptyset$ **do**
4: $LCT = \{j \in \mathcal{T}, t_j \leq CT,$ j is available$\}$
5: **if** $LCT \neq \emptyset$ **then**
6: $next \leftarrow$ task with highest priority from LCT
7: $\mathcal{T} \leftarrow \mathcal{T} - \{next\}$
8: $A(next) \leftarrow k$ ▷ Assign $next$ to workstation k
9: $CT \leftarrow CT - t_{next}$
10: **else**
11: $CT \leftarrow c$
12: **if** $k < m$ **then**
13: $k \leftarrow k + 1$ ▷ Open a workstation
14: **end if**
15: **end if**
16: **end while**
17: **return** A
18: **end function**

D. Neighborhood moves

Three neighborhood moves are considered in the current approach. The first is based on the insertion of tasks ($\mathcal{N}_{insertion}$), the second on the exchange of tasks ($\mathcal{N}_{swap}$), and the third on the increment of delay ($\mathcal{N}_{delay}$).

1) Insertion of task: Given a solution S and a task j:

- the nearest predecessor position of task j in $\mathcal{O}$ is denoted by $pred_{near}(j)$, and its workstation is $ws_{pred}(j)$. If task j has no predecessors, $pred_{near}(j) = 0$ and $ws_{pred}(j) = 1$.
- The nearest successor position of task j in $\mathcal{O}$ is denoted by $suc_{near}(j)$, and its workstation is $ws_{suc}(j)$. If task j has no successors, $suc_{near}(j) = n$ and $ws_{suc}(j) = m$.

A neighbor solution is obtained by removing task j from its current position and inserting it into the interval $[pred_{near}(j) + 1, suc_{near}(j) - 1]$ while assigning j to a workstation in the range $[ws_{prec}(j), ws_{suc}(j)]$. The delay of task j is reset to zero.

2) Swap of tasks: Given a solution S and a task j_1, compute $pred_{near}(j_1)$ and $suc_{near}(j_1)$ as described in Section III-D.1. Select a task $j_2 \in [pred_{near}(j_1) + 1, suc_{near}(j_1) - 1]$, such that $j_1 \in [pred_{near}(j_2) + 1, suc_{near}(j_2) - 1]$. A neighbor solution is obtained by swapping the position of j_1 and j_2 in vector $\mathcal{O}$ and exchanging their workstation assignments. The delays of tasks j_1 and j_2 are reset to zero.

3) Insertion of delay: Given a feasible solution S and a workstation k whose total processing time is less than the cycle time C, select a task j in workstation k. A neighbor solution is obtained by incrementing the delay of task j. The increment cannot make the last task of the workstation finish after the cycle time.

E. Evaluation function

The problem objective is to minimize peak power consumption while maintaining feasible solutions. $\mathcal{N}_{insertion}$ and $\mathcal{N}_{swap}$ respect the precedence constraints but not necessarily the cycle time constraint. As solutions violating the cycle time constraint could be generated, we use a penalty-based evaluation function. The penalty for solution s is computed as the cumulative excess time beyond the cycle time constraint across all workstations. Equation 4 formalizes this penalty calculation, where 'd_j' represents the delay of task j and c denotes the cycle time:

$$\text{penalty}(s) = \sum_{k \in \mathcal{M}} \max(0, \sum_{j \in S_k} (t_j + d_j) - c) \qquad (4)$$

The evaluation function combines solution quality and cycle time feasibility, as defined in Equation 5. Let:

- power_peak(s) be the peak power consumption of solution s
- penalty(s) be the penalty calculated using Equation 4
- W be a dynamic penalty weight sampled from a uniform distribution $U(0, w_{max})$, where 'w_{max}' represents the instance's maximum power consumption (Line 4).

$$f(s) = \text{power\_peak}(s) + \text{penalty}(s) \cdot W \qquad (5)$$

F. Local search

The local search procedure is used to improve generated solutions by systematically exploring three distinct neighborhood structures.

Employing a first-improvement strategy with randomized evaluation order, the algorithm examines neighbor solutions until finding an improving move. When no improvement exists for a given neighborhood (indicating a local optimum), all neighboring solutions are evaluated. The procedure terminates when the current solution is a local optimum for all three neighborhoods.

Algorithm 3 formalizes this local search process. The algorithm receives two inputs: a candidate solution S and an evaluation function $f(\cdot)$. The main loop (lines 3–19) continues while improvements exist. At line 4, the order in which $\mathcal{N}_{insertion}$ and $\mathcal{N}_{swap}$ will be applied is randomly chosen. At line 12, the neighborhood $\mathcal{N}_{delay}$, if the current solution is feasible. Lines 14 - 18 check whether the local search has found a local optimum for all neighborhoods.

TABLE II: Performance Comparison by Instance Group

Instance				#feasible solutions				gap.max [%]				gap.avg [%]				time.avg [s] (#iterations)		
group	tasks	machines	cycle	V1	V2	V3	V4	V1	V2	V3	V4	V1	V2	V3	V4	V1	V2	V3
MERTENS	7	2 - 6	6 - 20	100	100	100	90	**0.00**	**0.00**	5.10	47.46	**0.00**	**0.00**	0.31	5.27	9.58	7.10	8.77
BOWMAN8	8	3 - 7	17 - 37	100	100	100	85	**11.27**	**11.27**	15.49	88.46	**1.51**	**1.51**	1.93	19.77	13.36	10.31	13.23
JAESCHKE	9	4 - 8	6 - 13	100	100	100	97	**11.84**	**11.84**	14.47	47.14	1.65	**1.62**	3.02	8.22	13.43	9.44	12.51
JACKSON	11	2 - 6	9 - 30	100	100	100	78	**2.60**	**2.60**	3.06	61.22	0.99	**0.69**	1.21	14.96	25.76	18.15	24.46
MANSOOR	11	1 - 5	45 - 241	100	100	100	88	**24.07**	**24.07**	**24.07**	32.26	4.81	**4.70**	4.84	11.20	27.09	19.34	26.82
MITCHELL	21	3 - 7	16 - 46	100	100	100	65	**6.72**	**6.72**	**6.72**	46.15	2.45	**2.42**	2.42	17.42	89.19	59.76	88.01
ROSZIEG	25	5 - 9	16 - 34	100	100	100	65	**4.85**	**4.85**	**4.85**	34.95	1.96	**1.79**	2.10	14.68	136.78	90.94	136.31
HESKIA	28	2 - 6	171 - 666	100	100	100	55	6.71	**6.50**	6.71	60.34	2.90	**2.78**	2.94	28.79	788.15	630.49	790.27
BUXEY	29	8 - 12	28 - 54	100	100	100	52	**5.98**	**5.98**	**5.98**	27.95	**2.34**	2.43	2.48	14.33	346.71	273.87	344.47
SAWYER30	30	8 - 12	28 - 54	100	100	100	50	**7.74**	12.90	8.14	34.78	**3.14**	3.34	3.21	17.25	415.70	347.46	414.88
GUNTHER	35	8 - 12	44 - 82	100	100	100	58	**5.00**	**5.00**	**5.00**	65.15	2.21	**2.05**	2.13	13.66	485.36	369.65	485.38
WARNECKE	58	17 - 21	99 - 120	50	50	50	50	1.88	**1.61**	1.72	27.11	0.88	**0.81**	0.86	7.70	(6998)	(6216)	(6997)
LUTZ2	89	22 - 26	20 - 30	85	84	88	50	5.15	4.32	**3.34**	25.64	0.93	**0.85**	0.93	6.80	(23708)	625.27	(24913)
			(sum/max/avg)	1235	1234	**1238**	883	**24.07**	**24.07**	**24.07**	88.46	1.98	**1.92**	2.18	13.85	334.71	**266.29**	334.25

Algorithm 3 Local-search algorithm

function LOCAL_SEARCH($S, f(\cdot)$)
  $eval\_begin \leftarrow f(S)$, $is\_local\_optimum \leftarrow False$
  while $is\_local\_optimum \neq True$ **do**
    if $x > 0.5 \mid x \sim \mathcal{U}(0,1)$ **then**
      $S \leftarrow first\_improvement(S, \mathcal{N}_{swap})$
      $S \leftarrow first\_improvement(S, \mathcal{N}_{insertion})$
    else
      $S \leftarrow first\_improvement(S, \mathcal{N}_{insertion})$
      $S \leftarrow first\_improvement(S, \mathcal{N}_{swap})$
    end if
    if S is feasible **then**
      $S \leftarrow first\_improvement(S, \mathcal{N}_{delay})$
    end if
    if $f(S) = eval\_begin$ **then**
      $is\_local\_optimum \leftarrow True$
    else
      $eval\_begin \leftarrow f(S)$
    end if
  end while
  return S
end function

IV. COMPUTATIONAL EXPERIMENTS

To analyze the dependency of our proposed algorithm on its components, we evaluated four variants:

- **V1**: Baseline variant as described in Section III
- **V2**: Variant without the swap move to assess its impact
- **V3**: Variant with a fixed penalty factor (set to the maximum energy consumption)
- **V4**: Single-iteration variant to evaluate the benefit of restarts

We evaluated our approach using an extended benchmark[1] from previous research. The benchmark comprises:

- **Normal instances**: 65 instances (5 per family) with fixed workstation count m and cycle time c computed via SALBP-2 solver
- **Extended instances**: 65 instances (5 per family) with relaxed cycle time $c' = \lceil 1.3 \cdot c \rceil$

Task power consumption values w_j were generated from a uniform distribution $U(5, 50)$ following previous work [6], [7], [8].

[1] https://github.com/ZZFreya/SALB3PM/tree/main/benchmarks/1993

The implementation used GNU C++ 9.4.0 with `-O3` and `-march=native` optimizations. All experiments ran on an AMD EPYC 7452 32-Core Processor clocked up to 3.5 GHz running Ubuntu 20.04.2 LTS. A run terminates when either of the following criteria is met: (1) a 1000 seconds execution time limit or (2) completion of 50000 iterations. Each variant was executed 10 times per instance.

For each instance, we define:

- **Best Known Solution (BKS)**: The optimal power peak when known (obtained via the MaxSAT formulation from [8]), otherwise the minimum observed across all experimental runs
- **Relative Gap**: $\text{gap}(s) = 100 \times \frac{f(S) - \text{BKS}}{\text{BKS}}$ for solution s

Table II presents results aggregated by instance groups, organized in five column blocks:

- **Instance Group**: Characteristics of instance groups
- **Feasible solutions**: Number of runs where a feasible solution was found
- **Maximum Gap**: Worst-case performance per group
- **Average Gap**: Mean performance across groups
- **Computational Performance**: Average runtime (with iteration counts for time-limited runs). V4 results are omitted due to negligible computation times.

TABLE III: Comparison considering all instances

Metrics	Variant			
	V1	**V2**	**V3**	**V4**
gap.max(%)	**24.07**	**24.07**	**24.07**	88.46
gap.avg(%)	2.04	**1.98**	2.25	13.48
#best	66	**69**	58	24
#best10	40	**43**	38	14
#feasible	**124**	**124**	**124**	102
#feasible10	**123**	**123**	**123**	78

The final row aggregates results from overall instance groups. Tables III and IV present results grouped by variant. Table III shows aggregate results for all instances, including:

- Maximum and average relative gaps
- Count of instances where each variant achieved:
  - The best solution in at least one run (#best)
  - The best solution in all runs (#best10)
  - At least one feasible solution (#feasible)
  - Feasible solutions in all runs (#feasible10)

Table IV focuses on the 83 instances with known optimal solutions, reporting:

- Maximum and average relative gaps
- Count and percentage of instances where optimal solutions were found:
  - In any run (#opt)
  - In all runs (#opt10)

TABLE IV: Comparison considering known optimal solutions

Metrics	Variant			
	V1	**V2**	**V3**	**V4**
gap.max(%)	**24.07**	**24.07**	**24.07**	88.46
gap.avg(%)	2.42	**2.38**	2.73	12.77
#opt	42	**43**	39	24
#opt (%)	50.60	**51.81**	46.99	28.92
#opt10	40	**42**	38	9
#opt10 (%)	48.19	**50.60**	45.78	10.84

Our experimental results show that variant V2 achieved marginally superior performance. As shown in Table III, V2 obtained the best solutions for 69 instances, representing a 4.55% improvement over other variants. Furthermore, V2 consistently found optimal solutions across all 10 runs for 43 instances (as indicated by the #best10 metric). Interestingly, while V3 produced feasible solutions in slightly more runs than V2 (Table II), V2 exhibited better computational efficiency, running 20.4% faster than V1 and V3.

The importance of the restart mechanism becomes evident when examining V4's performance. Table III shows that V4 produced 17.74% fewer feasible solutions than other variants. If we consider all runs, this reduction is up to 36.59%. For instances with known optimal solutions (Table IV), V4 achieved optimal solutions for only 28.92% of instances and performed at least 18.07% worse than other variants.

The adaptative mechanism improved the solution quality. V1 achieved a 2.04% average gap compared to V3's 2.25, and V1 found 13.79% more best solutions than V3. We identified some instances where all variants failed to find feasible solutions, like the normal instance 'LUTZ2-9' and all normal instances in the 'Warnecke' group. Additionally, for instance 'LUTZ2-7', some algorithm runs failed to produce feasible solutions.

V. CONCLUSIONS

This paper presents a metaheuristic approach for the Simple Assembly Line Balancing Problem with Power Peak Minimization (SALB3PM). Our proposed method combines a multi-start framework with three distinct neighborhood moves - task insertion, task swap, and delay incrementing - and a dynamic penalty mechanism. Through four designed variants, we analyze the algorithm's dependency on its key components, evaluating the relative contribution of swap movement, the effectiveness of the dynamic penalty scheme, and the value of multiple restarts in the search process. The experimental results demonstrate that the best-performing variant achieved an average optimality gap of 2.38% relative to known optimal solutions. This variant produced optimal solutions across all runs for more than 50% of instances with known optima. However, we found that the inclusion of swap movement had a negative impact on the solution quality.

As future work, our findings suggest the investigation of novel representations and neighborhood structures that simultaneously optimize task ordering and inter-task idle time. Besides this, the development of a memory-based mechanism to improve the search process. Another direction that we will explore is the hybridization with exact methods, particularly to solve challenging instances. A possibility of hybridization is the use of exact methods to compute optimal delays for given task assignments.

ACKNOWLEDGMENT

This study was supported in part by the International Research Center "Innovative Transportation and Production Systems".

REFERENCES

[1] I. Baybars, "A Survey of Exact Algorithms for the Simple Assembly Line Balancing Problem," *Management Science*, vol. 32, no. 8, pp. 909–932, Aug. 1986. [Online]. Available: https://pubsonline.informs.org/doi/10.1287/mnsc.32.8.909

[2] A. Scholl and S. Voss, "Simple assembly line balancing - Heuristic approaches," *Journal of Heuristics*, vol. 2, no. 3, pp. 217–244, 1997. [Online]. Available: http://link.springer.com/10.1007/BF00127358

[3] A. Scholl and C. Becker, "State-of-the-art exact and heuristic solution procedures for simple assembly line balancing," *European Journal of Operational Research*, vol. 168, no. 3, pp. 666–693, Feb. 2006. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0377221704004795

[4] M. Fathi, D. B. M. M. Fontes, M. Urenda Moris, and M. Ghobakhloo, "Assembly line balancing problem: A comparative evaluation of heuristics and a computational assessment of objectives," *Journal of Modelling in Management*, vol. 13, no. 2, pp. 455–474, May 2018. [Online]. Available: https://www.emerald.com/insight/content/doi/10.1108/JM2-03-2017-0027/full/html

[5] IEA (2024), "World Energy Outlook 2024," International Energy Association, Paris, Tech. Rep. Annex A, 2024. [Online]. Available: https://www.iea.org/reports/world-energy-outlook-2024

[6] P. Gianessi, X. Delorme, and O. Masmoudi, "Simple Assembly Line Balancing Problem with Power Peak Minimization," in *Advances in Production Management Systems. Production Management for the Factory of the Future*, F. Ameri, K. E. Stecke, G. von Cieminski, and D. Kiritsis, Eds., vol. 566. Austin, TX, United States: Springer International Publishing, Sep. 2019, pp. 239–247. [Online]. Available: {https://link.springer.com/10.1007/978-3-030-30000-5_31}

[7] M. Py, A. Tuyaba, L. Deroussi, N. Grangeon, and S. Norre, "Application of SAT to the Simple Assembly Line Balancing Problem with Power Peak Minimization," in *15th Pragmatics of SAT international workshop, a workshop of the 27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024)*, 2024. [Online]. Available: https://hal.science/hal-04778733

[8] Z. Zheng, S. Cherif, and R. S. Shibasaki, "Optimizing Power Peaks in Simple Assembly Line Balancing Through Maximum Satisfiability," in *2024 IEEE 36th International Conference on Tools with Artificial Intelligence (ICTAI)*, 2024, pp. 363–370.

[9] D. Lamy, X. Delorme, and P. Gianessi, "Line Balancing and Sequencing for Peak Power Minimization," *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 10 411–10 416, 2020. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S2405896320335448

[10] X. Delorme, P. Gianessi, and D. Lamy, "A new Decoder for Permutation-based Heuristics to Minimize Power Peak in the Assembly Line Balancing," *IFAC-PapersOnLine*, vol. 56, no. 2, pp. 3704–3709, 2023. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S2405896323019456

[11] R. Martí, R. Aceves, M. T. León, J. M. Moreno-Vega, and A. Duarte, "Intelligent Multi-Start Methods," in *Handbook of Metaheuristics*, M. Gendreau and J.-Y. Potvin, Eds. Cham: Springer International Publishing, 2019, pp. 221–243. [Online]. Available: {https://doi.org/10.1007/978-3-319-91086-4_7}