mathematics

MDPI

Article

A Simulated Annealing and Variable Neighborhood Search Hybrid for Sequencing Interrelated Activities

Gintaras Palubeckis [1,*], Alfonsas Misevičius [1] and Zvi Drezner [2]

[1] Faculty of Informatics, Kaunas University of Technology, Studentu 50, 51368 Kaunas, Lithuania; alfonsas.misevicius@ktu.lt
[2] College of Business and Economics, California State University–Fullerton, Fullerton, CA 92834, USA; zdrezner@fullerton.edu
* Correspondence: gintaras.palubeckis@ktu.lt

Abstract

Determining an appropriate sequence of interrelated activities is one of the keys to developing a complex product. One of the approaches used to sequence activities consists of solving the feedback length minimization problem (FLMP). Several metaheuristic algorithms for this problem have been reported in the literature. However, they suffer from high computational costs when dealing with large-scale problem instances. To address this research gap, we propose a fast hybrid heuristic for the FLMP, which integrates the simulated annealing (SA) technique with the variable neighborhood search (VNS) method. The local search component of VNS relies on a fast insertion neighborhood exploration procedure performing only $O(1)$ operations per move. Using rigorous statistical tests, we show that the SA-VNS hybrid is superior to both SA and VNS applied individually. We experimentally compare SA-VNS against the insertion-based simulated annealing (ISA) heuristic, which is the state-of-the-art algorithm for the FLMP. The results demonstrate the clear superiority of SA-VNS over ISA. The SA-VNS hybrid technique produces equally good or better results across all tested problem instances. In particular, SA-VNS is able to find better solutions than ISA on all instances of size 150 or more. Moreover, SA-VNS requires two orders of magnitude less CPU time than the ISA algorithm. Thus, SA-VNS achieves excellent performance regarding solution quality and running time.

Keywords: combinatorial optimization; heuristics; variable neighborhood search; simulated annealing; product development; design structure matrix

MSC: 90B80; 90C27; 90C59

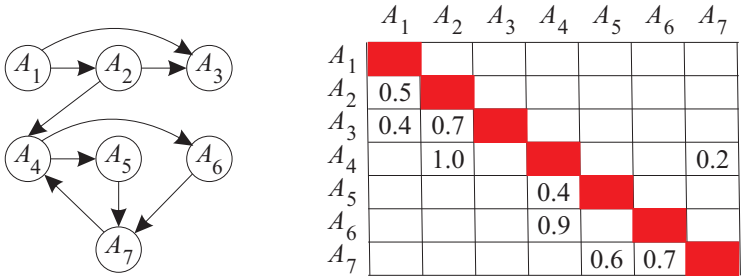
Academic Editor: Petr Stodola

Received: 27 November 2025
Revised: 7 January 2026
Accepted: 8 January 2026
Published: 12 January 2026

1. Introduction

Complex product development (PD) projects are typically conceptualized as a network of interconnected activities [1–5]. The activities interact by exchanging information during the PD process. The network of information flows in a PD project may contain activities that are mutually dependent, either directly or indirectly. Figure 1 shows an example of the network represented by a directed graph, in which vertices correspond to activities $A_1, \ldots, A_7$ and arcs indicate information flows. In this example, activities A_4, A_5, A_6, and A_7 are mutually dependent. They constitute a coupled set of design activities. Suppose that the vertices of the digraph are arranged in a sequence. Then, it may happen that an arc connects a downstream activity with an upstream activity. In such a case, the latter receives feedback information from the former. For the vertex ordering $A_1, \ldots, A_7$ in Figure 1, the

only feedback arc is (A_7, A_4). It is intuitively appealing that there is a causal relationship between the number of feedback arcs in a sequence of activities and the PD time and cost. The goal is to have the least amount of feedback information during the PD process.



(a)

	A_1	A_2	A_3	A_4	A_5	A_6	A_7
A_1							
A_2	0.5						
A_3	0.4	0.7					
A_4		1.0					0.2
A_5				0.4			
A_6				0.9			
A_7					0.6	0.7	

(b)

Figure 1. Activity digraph (**a**) and an example of the corresponding DSM (**b**).

A well-known approach for modeling and analyzing relations between activities is the design structure matrix (DSM) method introduced by Steward [6]. The DSM is a square matrix of dimension $n \times n$, where n is the number of activities. The off-diagonal entries of the matrix indicate the level of dependency of one activity upon another. An arc connecting vertex s with vertex q in the activity digraph is represented by the entry in row q and column s of the DSM. For example, as shown in Figure 1, the DSM entry corresponding to the arc (A_7, A_4) is 0.2. Steward [6] used a mark ("×") instead of numerical values to represent relations between activities. Such a matrix is referred to as binary DSM [7]. Eppinger et al. [7] proposed to use the concept of numerical DSM. Such DSMs allow representing both strong and weak precedence relations between activities. An example of the numerical DSM is shown in Figure 1b. Positive entries below the main diagonal represent forward dependencies for a given sequence of activities, and a single positive entry above the main diagonal represents a feedback dependency. In the figure, blank cells indicate no information dependency between activities. The DSMs were used in many studies devoted to PD process planning [8–16]. There are several reviews concerning DSM applications and extensions [17–19]. A bibliometric analysis of the DSM literature can be found in the paper of Piccirillo et al. [20].

An important optimization problem arising in the DSM context is to minimize feedback and its scope by sequencing the activities. Over the years, a variety of objective functions have been proposed in the literature to solve this problem [2]. Let us denote the level of dependency of activity q on activity s (the (q, s)th entry of the DSM) by a_{qs}. One natural objective function is the total feedback expressed as a sum of the superdiagonal entries of the DSM: $F_{\text{feedback}} = \sum_{q=1}^{n-1} \sum_{s=q+1}^{n} a_{qs}$. This objective function was suggested by Steward [6] for binary DSMs and was later used in studies addressing numerical DSMs [21,22]. Gebala and Eppinger [23] proposed an activity sequencing model in which the objective function is the total feedback length as follows:

$$F(p) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (j-i) a_{p(i)p(j)}, \tag{1}$$

where p is a permutation of activities ($p(i)$ and $p(j)$ denote the ith and jth activities in the sequence, respectively). Many researchers [2,7,24–26] have reported the significance of the feedback length objective function. Compared with F_{feedback}, the feedback length objective function (1) is better suited for reducing the PD process duration and cost [4]. The related

problem is called the feedback length minimization problem (FLMP for short). It can be stated as follows:

$$\min_{p\in\Pi} F(p) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (j-i) a_{p(i)p(j)}, \tag{2}$$

where Π is the set of all n-element permutations.

1.1. Related Work

As shown by Qian and Lin [25], the FLMP can be expressed as a quadratic binary program. These authors have also proposed two ways to linearize this program. They used CPLEX to solve the linear models (named LIP–1 and LIP–2 in [25]). Optimal solutions were obtained for FLMP instances of a size up to 14 activities. A different approach to solve the FLMP exactly was undertaken by Shang et al. [26]. They developed a parallel branch-and-prune algorithm for this problem. To accelerate the search process, the algorithm uses hash functions. Computational results have shown that their algorithm HAPBP (hash address-based parallel branch-and-prune) can find optimal solutions for problem instances with up to 25 interrelated activities.

As was observed by Qian and Lin [25], the FLMP is an NP-hard problem. Therefore, exact methods may be impractical due to problem size and computation time. To obtain good solutions for large FLMP instances, fast heuristic algorithms are needed. Several heuristic techniques have been proposed in the literature for tackling the problem. Altus et al. [27] developed a permutation-based genetic algorithm (GA) for solving (2). However, the authors provided very limited experimental results to support their approach. Another application of GA for (2) was reported by Whitfield et al. [28]. The focus of this paper was to determine the most efficient configuration of the algorithm. Meier et al. [2] proposed an ordering messy GA (OmeGA) hybridized with a local search procedure. The results of computational experiments have shown the effectiveness of this hybrid approach. Lancaster and Cheng [24] presented a fitness differential adaptive parameter control evolutionary algorithm (FDAPCEA) for solving the FLMP. The algorithm was tested on three FLMP benchmarks from the literature. Qian and Lin [25] developed an exchange-based local search heuristic (named EBH) for (2). However, like other local search algorithms, this heuristic can be trapped in a local minimum that is significantly worse than the global one. More recently, Lin et al. [29] proposed an insertion-based simulated annealing algorithm (ISA) and an insertion-based genetic algorithm (IGA) for the FLMP. Both algorithms heavily rely on an insertion-based heuristic, which is a kind of local search procedure. At each iteration of ISA, a swap move for a pair of activities is first performed a certain number of times, and then the obtained solution is improved using the insertion-based heuristic. In the case of IGA, this heuristic is applied to each individual of the initial population and to each offspring created by crossover. If the offspring are too similar to their parents, then an additional mutation step is executed. As experimentally shown by Lin et al. [29], the performances of ISA and IGA are incredibly similar regarding the quality of the solution produced and the time taken. Through the experimental results, it can be concluded that ISA and IGA are state-of-the-art algorithms for solving the FLMP. In a recent paper, Attari-Shendi et al. [5] stated that "the combination of the insertion-based heuristic with simulated annealing has led to an effective approach that outperforms existing approaches in large scales". Lately, Khanmirza et al. [30] have proposed an imperialist competitive algorithm (ICA) for feedback length minimization. They reported computational results on a small set of FLMP instances. The ICA found equally good or only slightly worse solutions compared with the ISA (see Table VII in [30]). On the other hand, ICA took less time to run than the ISA. Khanmirza et al. [30] also presented a genetic algorithm for the minimization of the feedback length in the DSM. This algorithm, however, showed inferior performance compared

with the ICA approach to the problem. In another paper, Yazdanjue and Khanmirza [31] addressed an optimization problem in which they combined two objective functions: one representing the feedback length, and the second one representing the degree of modularity in the modularized DSM. To solve this problem, the authors proposed a metaheuristic algorithm named discrete particle swarm optimization simulated annealing. They reported computational results for a set of small matrices. The main advantages and disadvantages of the different methods proposed in the literature are summarized in Table 1.

Table 1. Advantages and disadvantages of the existing algorithms for the FLMP.

Algorithm	Advantages	Disadvantages
LIP–1 [25]	Guarantees global optimality	Solves only small-scale instances
LIP–2 [25]	Guarantees global optimality	Solves only small-scale instances
HAPBP [26]	Guarantees global optimality	Particularly long computation time
GA [27]	Flexible in modifying the objective function	Long computation time
GA [28]	Efficient structure of the GA	Long computation time
OmeGA [2]	Uses local search strategies	Takes too long
FDAPCEA [24]	Adaptive parameter control in evolution strategy	Convergence is not fast
EBH [25]	Applies an iterative improvement scheme	Difficult to avoid falling into local optima
ISA [29]	Insertion-based cooling process	Difficult implementation; convergence speed is slow
IGA [29]	Applies insertion-based heuristic to each individual in the population	Difficult implementation; convergence speed is slow
GA [30]	Adaptive parameter tuning	Requires longer computation time
ICA [30]	Strong global search ability	Low degree of search intensification

The focus of this paper is on the development of an algorithm for the FLMP that combines simulated annealing (SA) and a variable neighborhood search (VNS) method. The SA component of the approach generates initial solutions for VNS. The latter is used as a powerful technique for search intensification. The choice of the VNS metaheuristic is motivated by its effectiveness, which is demonstrated through its use in developing algorithms for various combinatorial optimization problems. In the past few years, VNS algorithms have been proposed to solve a number of such problems, including the capacitated dispersion problem [32], truck routing within a naval port [33], minimum load coloring [34], integrated production and assembly in smart manufacturing [35], the obnoxious p-median problem [36], electric vehicle routing with simultaneous pickup and delivery [37], the vertex separator problem [38], swap-body vehicle routing [39], the periodic maintenance problem [40], and multi-depot vehicle routing [41]. The VNS metaheuristic was successfully applied to optimization problems whose solutions can be represented as permutations, such as in the case of the FLMP (such as no-delay single machine scheduling [42], the one-commodity pickup-and-delivery traveling salesman problem [43], multi-quays berth allocation and crane assignment [44], bandwidth reduction [45], and the traveling repairman problem with profits [46]).

The algorithm presented in the current paper combines the VNS methodology with the SA heuristic. In the literature, there exist several studies that have applied the VNS algorithm in combination with other approaches. Xiao et al. [47] proposed a VNS-based hybrid algorithm for solving capacitated vehicle routing problems. Their algorithm incorporates SA into the VNS framework. The key idea of this algorithm is to replace the local search step with the solution acceptance rule used in SA. Salehipour and Sepehri [48] presented a different combination of VNS and SA for the traveling repairman problem. Their algorithm uses VNS as a procedure to obtain solutions during the cooling process. Zhou et al. [49] addressed the problem of the joint optimization of storage allocation and order picking for fresh products. The authors developed a particle swarm-guided hybrid genetic-simulated annealing algorithm for solving this problem. They integrated SA with VNS and applied

this combination as a local search optimizer. Drezner et al. [50] combined VNS with GA to solve the planar p-median problem. They applied a distribution-based VNS in the final phase of the approach. Irawan et al. [51] proposed a hybrid algorithm for the continuous location and maintenance routing problem for offshore wind farms. In this algorithm, the VNS is embedded in the GA as a functional part. Two versions of the VNS algorithm are used: mini VNS and full VNS. Kocatürk et al. [52] developed an algorithm to address the multi-depot heterogeneous vehicle routing problem with backhauls. Their algorithm integrates the greedy randomized adaptive search procedure (GRASP) and the VNS method. The latter is applied in the local search step of the GRASP. Kong et al. [53] presented a hybrid algorithm for the parallel-batching machine scheduling problem in the unrelated parallel machines environment. They combined a shuffle frog leap algorithm and the VNS technique. In this implementation, VNS is used as a local optimizer. Amaldass et al. [54] developed an algorithm based on VNS and ant colony optimization (ACO) to solve the vector job scheduling problem. The ACO algorithm was used to generate an initial solution for VNS. Simeonova et al. [55] presented a version of VNS for a real-life vehicle routing problem. They hybridized VNS with an adaptive memory procedure. Bouzid et al. [56] proposed a VNS-based heuristic to solve the capacitated vehicle routing problem. The authors integrated a Lagrangian relaxation method, called Lagrangian split, into the VNS scheme. Irawan et al. [57] developed a hybrid algorithm for large unconditional and conditional vertex p-center problems. The algorithm combines the VNS metaheuristic with an exact method. Daza-Escorcia and Álvarez-Martínez [58] introduced a static bike-sharing repositioning problem. To solve it, they proposed a matheuristic procedure combining VNS and integer linear programming techniques. Lan et al. [59] presented an effective hybrid approach to tackle the physician scheduling problem with multiple types of tasks. The approach incorporates VNS and a dynamic programming algorithm.

1.2. Contributions of This Paper

Analysis of the literature shows that there has been considerable interest in developing algorithms for the FLMP. However, these algorithms have been shown to work well for sequencing a relatively small number of interrelated activities. There is a lack of metaheuristic-based approaches capable of producing high-quality solutions for large-scale problem instances. Furthermore, it can be noted that the best existing algorithms are not sufficiently fast. Considering these observations, our motivation is to develop a reasonably fast heuristic algorithm that could perform well on large FLMP instances with several hundreds of activities. The aim of this paper consists of developing a metaheuristic-based algorithm that is capable of producing better solutions in shorter CPU times compared with the ISA algorithm of Lin et al. [29]. Our algorithm is constructed by combining the simulated annealing technique and a variable neighborhood search method. The choice of this combination was motivated by the good results of applying an SA and VNS hybrid for some other optimization problems on permutations, e.g., the profile minimization problem [60] and the bipartite graph crossing minimization problem [61]. To fully capture the benefits of both algorithms, we apply them iteratively. One of the main strengths of SA is its ability to approach global optimality. Meanwhile, the VNS algorithm employs a relatively strong mechanism to intensify the search near local optima. We exploit these strengths by combining the two algorithms. Of course, SA and VNS can be used individually. We experimentally investigated this scenario and compared the SA and VNS algorithms against their hybrid. One particular problem is developing a fast local search (LS) procedure for the FLMP. In this paper, we present an LS procedure that is based on performing activity insertion moves. In each iteration of LS, the entire insertion neighborhood of the current solution is explored in $O(n^2)$ time, which means that the procedure performs only $O(1)$

operations per move. The developed SA and VNS hybrid was tested on several FLMP benchmarks and a set of randomly generated DSMs. These problem instances contain up to 500 interrelated activities.

The main contributions of this paper are as follows:

- Hybridization of VNS with SA for solving the FLMP;
- A fast local search procedure for the FLMP;
- Numerical experimentation on FLMP instances of a size up to 500 interrelated activities to validate the effectiveness of the approach;
- Confirmation of the superiority of the hybrid algorithm over SA and VNS when they are used individually.

The rest of this paper is organized as follows. In the next section, we show how SA and VNS are integrated into a single framework. In Sections 3 and 4, we present the SA and VNS algorithms for the FLMP, respectively. Experimental results of the proposed approach and comparisons with the state-of-the-art are reported in Section 5. Discussion and concluding remarks are given in Sections 6 and 7, respectively.

2. Integrating SA and VNS

Simulated annealing and variable neighborhood search are two well-known metaheuristic techniques. Each of them was successfully applied to a number of very different optimization problems. SA is famous for its strategy of avoiding becoming stuck in local optima, and VNS provides a high level of intensification of the search process. Our idea is to combine the advantages of these two metaheuristics. We apply them in an iterative fashion. Each iteration is started by executing the SA algorithm. The solution returned is then further optimized using the VNS component of the approach. As is well known [62,63], the general strategy of the VNS algorithm is to explore the neighborhoods of the currently best found solution. If the latter is good enough already at the start of VNS, significant computation time savings can be achieved. Thus, owing to the fact that SA produces solutions of high quality, it is advantageous to run SA first and VNS second.

As mentioned before, the two components of the approach are applied iteratively. Each iteration begins by generating a random initial permutation of activities, which defines the starting state for the SA cooling process. The solution produced by SA is then passed to the VNS algorithm for potential further improvement. The iterations stop once the elapsed time exceeds a predefined timeout value. As is well known, the SA algorithm terminates when the temperature reaches a specified value close to zero. Without imposing a time limit, however, a straightforward implementation of the VNS algorithm can run arbitrarily long. To address this issue, we divide the overall CPU time limit into two parts: one allocated to SA and the other to VNS. If the time required for a single SA execution is known, the expected number of SA invocations during the iterative process can be estimated. To ensure the same number of VNS calls, the time limit for a single VNS run is then computed accordingly.

The proposed SA-VNS framework is designed to balance diversification and intensification within a fixed computational budget. Simulated annealing is used to generate diverse, high-quality starting solutions, while variable neighborhood search exploits these solutions to achieve further improvements. Since VNS may require unpredictable and potentially excessive running times, a time-controlled iterative scheme is adopted. The overall CPU time limit is therefore allocated between SA and VNS in a structured way, allowing multiple SA-VNS cycles to be executed while ensuring fairness and predictability across iterations. This strategy enables an effective exploration-exploitation trade-off and provides a clear, reproducible termination mechanism. Further details are provided in the pseudocode of the main part of the algorithm and its accompanying description.

Algorithm 1 shows the top-level pseudocode of our approach, denoted as SA-VNS. The algorithm iterates through the while-loop in Lines 2–12 until a stopping criterion is met. In our implementation of SA-VNS, the loop terminates when the maximum time limit t_{lim} is reached. Throughout the search, the algorithm memorizes both the best solution found in the current iteration and the globally best solution obtained thus far. They are denoted by $\tilde{p}$ and p^*, respectively, and their objective function values are, respectively, $\tilde{f}$ and f^*. In Line 3 of the pseudocode, a random permutation is generated, and in Line 4, our SA procedure is invoked. It returns solution $\tilde{p}$ and its value $\tilde{f}$, which are later passed to the VNS algorithm (Line 10). After the first execution of SA, the cut-off time for each run of VNS, denoted as $t_{\text{run\_VNS}}$, is computed. To share the CPU time resources, we use a parameter μ that ranges from 0 to 1. The running time limit for SA is set to $t_{\text{lim\_SA}} = \mu t_{\text{lim}}$; for VNS, it is set to $t_{\text{lim\_VNS}} = (1-\mu)t_{\text{lim}}$. Let $t_{\text{SA\_it}}$ denote the time taken by SA in the first SA-VNS iteration. Having $t_{\text{lim\_SA}}$ and $t_{\text{SA\_it}}$, we can calculate the predicted number of SA-VNS iterations using the formula $\lambda = \lceil t_{\text{lim\_SA}}/t_{\text{SA\_it}} \rceil$. Then, $t_{\text{run\_VNS}}$ is set to $t_{\text{lim\_VNS}}/\lambda$ (Line 5 of the pseudocode). If a predetermined amount of time expires during the execution of SA, then the SA-VNS iteration is stopped prematurely (Lines 7 and 8). Otherwise, the algorithm tries to improve the solution $\tilde{p}$ by making a call to the VNS procedure (Line 10). If the obtained permutation $\tilde{p}$ is better than p^*, then the globally best solution and its value are updated (Line 11).

Algorithm 1 Top-level description of SA-VNS.

```
    SA-VNS
1:  f* := ∞
2:  while time limit t_lim not reached do
3:     Randomly generate a permutation p̃ ∈ Π
4:     Apply SA(p̃, f̃, t_lim)
5:     if f* = ∞ then Compute t_run_VNS end if
6:     if t_lim expired then
7:        if f̃ < f* then p* := p̃, f* := f̃ end if
8:        Stop with the solution p* of value f*
9:     end if
10:    Apply VNS(p̃, f̃, t_lim, t_run_VNS)
11:    if f̃ < f* then p* := p̃, f* := f̃ end if
12: end while
13: Stop with the solution p* of value f*
```

We have described an implementation of SA-VNS in which a CPU time-based stopping rule is used. The algorithm has to be slightly modified when other termination criteria are adopted, such as, for example, the maximum number of calls to SA and VNS.

3. Simulated Annealing

In this section, we present our implementation of the SA method for the FLMP. The concept of simulated annealing is inspired by an analogy between the metallurgical process of annealing in thermodynamics and an optimization problem. This analogy was first perceived and exploited for solving combinatorial optimization problems by Kirkpatrick et al. [64] and Černý [65].

Simulated annealing is a stochastic local search method capable of escaping the local minimum by accepting worsening moves with a certain probability. When implementing SA, it is important to define how to move from one solution to a neighboring one. In our SA algorithm, insertion moves are used for this purpose. Given a permutation p on the set of activities, an insertion move removes an activity q from its current position in p and

inserts it at a different position. The set of all permutations that can be obtained from p by applying this operation is called the insertion neighborhood of p. We denote it by $N(p)$. Suppose that the current position of q is k and the target position is l; then let p' denote the solution obtained from p by relocating activity q from position k to position l. The resulting change in cost $\Delta(p, p') = F(p') - F(p)$ is called the move gain. First, let us consider the case where $k < l$. One can interpret the insertion move as performing $l - k$ times a pairwise exchange operation in which the activity q is interchanged with its current right neighbor in the permutation p. After the ith step of this process, the activity q appears to be temporarily placed in position $k + i$. Let the change in the objective function value with this step be denoted by Δ_{k+i}. Thus, $\Delta(p, p') = \sum_{m=k+1}^{l} \Delta_m$. A step of the insertion move is illustrated in Figure 2, where $k \leqslant 3$, $l \geqslant 4$, $m = 4$, and s is the right neighbor of q. In this figure, the arc for a DSM entry a_{vw} is directed from v to w. To efficiently compute the gain $\Delta(p, p')$, we maintain arrays $L = (L(1), \ldots, L(n))$ and $R = (R(1), \ldots, R(n))$, which are defined as follows: Let Q denote the set of activities. Suppose $q \in Q$ is such that $p(k) = q$. Then, $L(q) = \sum_{i=1}^{k-1} a_{p(i)q}$ and $R(q) = \sum_{i=k+1}^{n} a_{qp(i)}$. In our example, the caption of Figure 2 lists the entries of L and R for the activities q and s. The objective function (1) can be rewritten as $F(p) = \sum_{u=1}^{n-1} C_u(p)$, where $C_u(p) = \sum_{i=1}^{u} \sum_{j=u+1}^{n} a_{p(i)p(j)}$. The sums $C_u(p)$ can be calculated recursively as

$$C_u(p) = C_{u-1}(p) + R(p(u)) - L(p(u)), \tag{3}$$

assuming by convention that $C_0(p) = 0$. Let us continue to assume that $k < l$. When performing an insertion operation, the activity q is moved from left to right. Accordingly, the corresponding entry of the array L, as well as R, is changing. First, consider the case where $l = k + 1$, and then denote by $s = p(l)$ the right neighbor of $q = p(k)$. Then, it is easy to see that $C_u(p') = C_u(p)$ for $u = 1, \ldots, k-1, k+1, \ldots, n-1$. Thus, $\Delta(p, p') = F(p') - F(p) = C_k(p') - C_k(p)$. From (3), we find that $C_k(p) = C_{k-1}(p) + R(q) - L(q)$ and $C_k(p') = C_{k-1}(p') + R(s) + a_{sq} - (L(s) - a_{qs})$. Consequently,

$$\Delta(p, p') = \Delta_l = R(s) - L(s) + L(q) + a_{sq} - (R(q) - a_{qs}). \tag{4}$$

It can be observed that $L(q) + a_{sq}$ is the value of $L(q)$ in the permutation p'. Similarly, $R(q) - a_{qs}$ is the value of $R(q)$ in the same permutation. To represent the current $L(q)$ and $R(q)$ values while computing Δ_m for all $m \in \{k+1, \ldots, l\}$, we use auxiliary variables x and y. At the beginning, they are initialized with $L(q)$ and $R(q)$, respectively. The step gains Δ_m, $m = k+1, \ldots, l$, are calculated incrementally by first setting

$$x = x + a_{sq}, y = y - a_{qs} \tag{5}$$

and then using the following equation reformulated from (4):

$$\Delta_m = R(s) - L(s) + x - y, \tag{6}$$

where $s = p(m)$. In our example in Figure 2, initially $x = L(q) = 3$ and $y = R(q) = 12$. Applying (5) and (6), we obtain the following: $x = x + a_{sq} = 3 + 3 = 6$, $y = y - a_{qs} = 12 - 5 = 7$, and $\Delta_m = \Delta_4 = 2 - 8 + 6 - 7 = -7$.

Suppose now that $k > l$. In this case, initially, $x = R(q)$ and $y = L(q)$. Additionally, the computation of Δ_m is slightly different. The analogs of (5) and (6) take the following form:

$$x = x + a_{qs}, y = y - a_{sq}, \tag{7}$$

$$\Delta_m = L(s) - R(s) + x - y, \tag{8}$$

where $s = p(m)$ as before. The step gain Δ_m is computed for decreasing values of m, starting with $m = k - 1$.

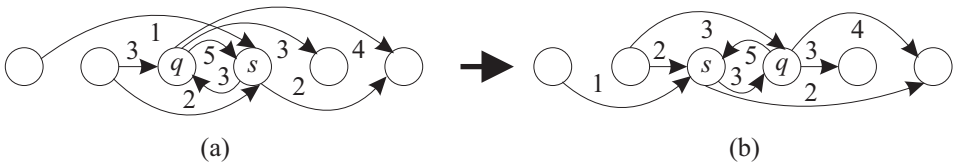


Figure 2. Swapping two adjacent activities: (**a**) $L(q) = 3$, $R(q) = 12$, $L(s) = 8$, and $R(s) = 2$ before move; and (**b**) $L(q) = 6$, $R(q) = 7$, $L(s) = 3$, and $R(s) = 5$ after move.

The pseudocode of computing $\Delta(p, p')$ is shown in Algorithm 2. The activity q is moved from position k in the permutation p to position l. Loops 4–10 compute the gain Δ when activity q is moved to the right, and Loops 12–18 compute Δ when q is moved to the left. The gain Δ is obtained by summing up the step gains Δ_m, which are computed either by (6) or (8). In the pseudocode, s stands for the closest neighbor of q in the moving direction. The arrays L and R are initialized and updated outside of the procedure `get_gain`. Clearly, the time complexity of this procedure is $O(n)$.

Algorithm 2 Computing the insertion gain.

```
    get_gain(q, l, p)
 1: k := p^{-1}(q) // p^{-1} is the inverse permutation of p
 2: Δ := 0
 3: if k < l then
 4:    x := L(q)
 5:    y := R(q)
 6:    for m = k + 1, ..., l do
 7:       s := p(m)
 8:       Update x, y by (5) and compute Δ_m by (6)
 9:       Δ := Δ + Δ_m
10:    end for
11: else
12:    x := R(q)
13:    y := L(q)
14:    for m = k − 1 to l by −1 do
15:       s := p(m)
16:       Update x, y by (7) and compute Δ_m by (8)
17:       Δ := Δ + Δ_m
18:    end for
19: end if
20: return Δ
```

Having the gain computation procedure, we can present the main part of our SA algorithm. Its pseudocode is given in Algorithm 3. The input to SA includes three parameters: the cooling factor α, the final temperature $T_{\min}$, and the number of moves I to be attempted at each temperature level. The initial temperature $T_{\max}$ is computed as $T_{\max} = \max_{p' \in N'} |\Delta(p, p')|$, where p is a starting random permutation generated in Line 3 of Algorithm 1, and N' is a sample of permutations chosen randomly from the insertion neighborhood $N(p)$. Using temperatures $T_{\max}$, $T_{\min}$, and factor α, the number of temperature levels $\bar{\gamma}$ can be calculated (Line 6). Both $T_{\max}$ and $\bar{\gamma}$ are computed during the first SA-VNS iteration and reused in subsequent iterations. Before starting the annealing process, the arrays L and R are initialized (Line 3). This step takes $O(n^2)$ operations. The arrays L and R are updated by the procedure `insert` (Line 15). The main part of SA is a double-nested loop (Lines 9–21). Inside the inner loop, the algorithm calls `get_gain`

and, if the move is accepted, triggers the procedure `insert` (Line 15). Provided $f < \tilde{f}$, the best solution is updated in Line 16. The outer loop of SA is equipped with a time-based stop condition to terminate the search prematurely. If this condition fails, the temperature is gradually reduced by a geometric cooling schedule $T := \alpha T$ (Line 20) until the final temperature $T_{\min}$ is reached.

Algorithm 3 Simulated annealing.

```
    SA(p̃, f̃, t_lim)
// α, T_min and I are parameters of the algorithm
 1: p := p̃
 2: f̃ = f := F(p)
 3: Construct arrays L and R
 4: if first call to SA then
 5:    Compute T_max
 6:    γ̄ := ⌊(log(T_min) − log(T_max)) / log α⌋
 7: end if
 8: T := T_max
 9: for γ = 1, ..., γ̄ do
10:    for i = 1, ..., I do
11:       Pick activity q and its new position l at random
12:       Δ:=get_gain(q, l, p)
13:       if Δ ⩽ 0 or exp(−Δ/T) ⩾ random(0, 1) then
14:          f := f + Δ
15:          Update p, L and R by invoking insert(q, l, p)
16:          if f < f̃ then p̃ := p, f̃ := f end if
17:       end if
18:    end for
19:    if elapsed time ⩾ t_lim then return
20:    T := αT
21: end for
22: return
```

Algorithm 4 gives the pseudocode of the procedure `insert`. The insertion operation is accompanied by updating the arrays L and R. The current position of activity q in the permutation p is denoted by k. The entries of L and R are updated for all activities placed in the positions of p between k and l inclusively. At the end, the procedure assigns q to the target position l (Line 17).

Algorithm 4 Performing an insertion move.

```
    insert(q, l, p)
 1: k := p^{-1}(q) // p^{-1} is the inverse permutation of p
 2: if k < l then
 3:    for m = k + 1, ..., l do
 4:       s := p(m)
 5:       Add a_sq to both R(s) and L(q)
 6:       Subtract a_qs from both L(s) and R(q)
 7:       p(m − 1) := s
 8:    end for
 9: else
10:    for m = k − 1 to l by −1 do
11:       s := p(m)
12:       Add a_qs to both L(s) and R(q)
13:       Subtract a_sq from both R(s) and L(q)
14:       p(m + 1) := s
15:    end for
16: end if
17: p(l) := q
18: return
```

From Algorithm 4, it is obvious that the running time of `insert` is linear in n. As already remarked, the same is true for the procedure `get_gain`. Then, the computational complexity of SA can be evaluated as $O(n\bar{\gamma}I)$. Typically, in SA algorithms, I is within a constant factor of n. In such a case, the complexity expression simplifies to $O(n^2\bar{\gamma})$, where $\bar{\gamma}$ is the number of temperature levels. This number depends on the initial temperature (calculated in Line 5 of Algorithm 3) and the SA parameters α and $T_{\min}$.

4. Variable Neighborhood Search

One of the weaknesses of SA is that it is always possible to become stuck at a local minimum that could be significantly worse than the global minimum. Therefore, if the time limit permits, we try to improve the solution returned by SA by applying the variable neighborhood search procedure. The VNS method is a metaheuristic technique introduced by Mladenović and Hansen [66] for solving optimization problems. The basic principle of VNS is to iteratively apply a neighborhood change mechanism and a local search procedure.

For better readability, we first provide a brief high-level overview of the proposed VNS approach. The algorithm iteratively explores the space of permutations by alternating between diversification and intensification phases. Diversification is achieved through a shaking mechanism that perturbs the current best solution (permutation), while intensification is performed via a local search that refines solutions toward local minima. This structure enables an effective balance between exploration of the solution space and exploitation of high-quality regions.

In our implementation of VNS, we use neighborhood structures that are appropriate for permutation-based combinatorial optimization problems. Given a permutation $p \in \Pi$, we define the neighborhood $\tilde{N}_r(p)$, $r \in \{1, \ldots, r_{\max}\}$, as a set of all permutations that can be obtained from p by performing r pairwise interchanges of activities subject to the condition that no activity is moved more than once.

The pseudocode of the VNS algorithm for the FLMP is presented in Algorithm 5. As mentioned previously, VNS starts from the solution produced by the SA method. The input to VNS includes parameter $r_{\min}$, which defines the size of the neighborhood each iteration of the search is started from. At the beginning of each iteration, the maximum possible size of the neighborhood, $r_{\max}$, and the size change step r_{step} are calculated. The value of $r_{\max}$ is an integer number drawn uniformly at random from the range $[\eta_1 n, \eta_2 n]$, where η_1 and $\eta_2 \geqslant \eta_1$ are parameters of the algorithm. To define r_{step}, we use the scaling factor $\beta > 0$. We set r_{step} to $\max(\lfloor r_{\max}/\beta \rfloor, 1)$. The inner loop of the algorithm iteratively applies three procedures: (a) `shake` (Algorithm 6), which generates a solution $p \in \tilde{N}_r(\tilde{p})$ by performing r pairwise interchanges of activities; (b) LS, which returns a locally optimal solution p; and (c) `neighborhood_change` (Algorithm 7), which chooses the neighborhood structure to continue the search and memorizes the best solution found thus far.

Algorithm 5 Variable neighborhood search.

VNS($\tilde{p}, \tilde{f}, t_{\text{lim}}, t_{\text{run\_VNS}}$)
// $r_{\min}$ is a parameter of the algorithm
1: $p := \tilde{p}$
2: $f :=$LS(p)
3: **if** $f < \tilde{f}$ **then** $\tilde{p} := p$, $\tilde{f} := f$ **end if**
4: **while** time limit for VNS run, $t_{\text{run\_VNS}}$, not reached **do**
5: $r := r_{\min}$
6: Compute $r_{\max}$ and r_{step}
7: **while** $r \leqslant r_{\max}$ **do**
8: $p :=$`shake`($\tilde{p}, r$)
9: $f :=$LS(p)
10: $r :=$`neighborhood_change`($p, \tilde{p}, f, \tilde{f}, r, r_{\min}, r_{\text{step}}$)
11: **if** t_{lim} expired **then return end if**
12: **end while**
13: **end while**
14: **return**

Algorithm 6 Shake function.

```
shake(p̃, r)
p := p̃
U := {1, ..., n}
for r times do
    Randomly select activities q, s ∈ U
    Swap positions of q and s in p
    U := U \ {q, s}
end for
return p
```

The local search procedure is at the heart of the VNS metaheuristic. Our implementation of this procedure, denoted as LS, is based on the exploration of the insertion neighborhood. At a high level, the proposed local search explores the solution space by systematically relocating each activity within the current permutation. For every iteration, all possible insertion positions of each activity are evaluated to identify the move that yields the greatest improvement in the objective function value. The search proceeds by repeatedly applying the best improving insertion move until no further improvement is possible, at which point a locally optimal solution is reached. Since each activity has $n-1$ possible relocation positions in the permutation, a total of $n(n-1)$ solutions in the insertion neighborhood are examined at each iteration.

Algorithm 7 Neighborhood change function.

```
neighborhood_change(p, p̃, f, f̃, r, r_min, r_step)
if f < f̃ then
    p̃ := p
    f̃ := f
    r := r_min
else
    r := r + r_step
end if
return r
```

The pseudocode of LS is given in Algorithm 8. The procedure starts by initializing the arrays L and R, which are used to calculate the gain of performing an insertion move. Then, for each activity q (taken in Line 6 of the pseudocode), two loops are executed (Lines 9–17 and 20–28). The gain Δ inside these loops is computed in exactly the same way as in the procedure `get_gain`. To keep track of the current best move, LS maintains a triplet (Δ^*, q^*, l) consisting of the best gain Δ^*, the activity q^* for which this gain is achieved, and the new position l for q^* in the current permutation. If $\Delta^* < 0$ after processing all activities, then the procedure `insert` (given in Algorithm 4) is triggered. We conclude this section with an observation regarding the complexity of LS. From the pseudocode in Algorithm 8, it is clear that an iteration of LS (Lines 4–33) takes $O(n^2)$ time. Indeed, first, all the statements within the inner loops (including those in Lines 11 and 22) execute a constant number of iterations. Second, from Algorithm 4, it can be seen that the complexity of `insert` is $O(n)$.

Algorithm 8 Local search.

```
LS(p)
Construct arrays L and R and compute f = F(p)
Δ* := −1
while Δ* < 0 do
  Δ* := 0
  for k = 1, ..., n do
    q := p(k)
    Δ := 0
    x := L(q), y := R(q)
    for m = k + 1, ..., n do
      s := p(m)
      Update x and y using (5) and compute Δ_m using (6)
      Δ := Δ + Δ_m
      if Δ < Δ* then
        Δ* := Δ
        q* := q, l := m
      end if
    end for
    Δ := 0
    x := R(q), y := L(q)
    for m = k − 1 to 1 by −1 do
      s := p(m)
      Update x and y using (7) and compute Δ_m using (8)
      Δ := Δ + Δ_m
      if Δ < Δ* then
        Δ* := Δ
        q* := q, l := m
      end if
    end for
  end for
  if Δ* < 0 then
    insert(q*, l, p)
    f := f + Δ*
  end if
end while
return f
```

5. Computational Experiments

In this section, we report on the results of computational tests to assess the performance of the proposed simulated annealing and variable neighborhood search hybrid for solving the FLMP. The effectiveness of the approach is evaluated by conducting two major experiments. In the first experiment, we compare SA-VNS against the insertion-based simulated annealing algorithm proposed by Lin et al. [29]. This algorithm is a state-of-the-art method for the FLMP. In the second experiment, we compare the SA-VNS algorithm with its components, SA and VNS, on a set of large-scale FLMP instances. In addition, we carry out an experiment aimed at comparing the running time between our LS procedure and the local search algorithm of Lin et al. [29].

5.1. Experimental Setup

The algorithm described in the preceding sections was coded in the C++ programming language. For the sake of comparison, we also coded the insertion-based simulated annealing algorithm of Lin et al. [29]. The experiments were performed on a laptop with an Intel Core i5-6200U CPU running at 2.30 GHz.

We ran the algorithms on the seven benchmark instances from the literature, as well as on a set of randomly generated FLMP instances. We used the following benchmark DSMs that were considered in previous studies: `steward'81` [6], `kusiak'91` [67], `austin'96` [68], and `qian'14` [25]. In addition, we tested the algorithms on the three DSMs from the book of Eppinger and Browning [4] as follows:

- `microprocessor` (this matrix represents Intel's existing product development process in 1992);
- `strategy` (this matrix models the research program strategy development process in Meat & Livestock Australia Ltd.);
- `automobile` (this DSM represents Ford's baseline hood development process in 1999).

However, the above-listed benchmark matrices are sparse and relatively small. The FLMP with these matrices is particularly easy to solve by metaheuristic algorithms, like those presented in [29]. We, therefore, randomly generated 60 extra matrices of dimensions ranging from 30 to 500. We adopted the same generation method as in [29]. The nonzero entries of each matrix are drawn uniformly at random from the interval $(0, 1)$. The matrix density is set to one of the following values: 10%, 20%, 30%, 40%, or 50%. The new matrices, as well as the source code of our algorithm, have been made publicly available (https://drive.google.com/drive/folders/1md2vyj3nQcwo1EakmJwrQT7JNyD4W2oJ?usp=sharing (accessed on 07 January 2026)).

Table 2 shows the maximum CPU time allowed for a run of the SA-VNS algorithm on random FLMP instances. For benchmark DSMs, SA-VNS was stopped after 1 s of CPU time. To be able to apply the same cut-off times for the insertion-based simulated annealing algorithm of Lin et al. [29], we implemented it as a multi-start procedure. We, however, allowed completing the first start of this algorithm, even if the maximum allotted time had elapsed. The performance of the algorithms was measured by the objective function value of the best solution found during the search, the average objective function value, and the average time taken to find the best solution in a run.

Table 2. CPU time limits for random instances.

n	Seconds
30	3
60	3
100	20
150	120
200	300
250	600
300	900
350	1200
400	1800
500	3600

5.2. Parameter Settings

The main parameters of our simulated annealing implementation were the cooling factor α and the number of iterations I, which were executed at each temperature. Based on the recommendations from the SA literature [69,70], we used the following values for these parameters: $\alpha = 0.95$ and $I = 100n$.

Our VNS algorithm was configured by setting the values of parameters $r_{\min}$, η_1, and η_2, which define the minimum and maximum size of the neighborhood, and the parameter β, which controls switching to the next neighborhood. We conducted some experiments to find good settings for the VNS parameters. We examined the performance of SA-VNS on a training dataset consisting of five problem instances of size 100 and five instances of size 200. These instances were generated randomly using the same procedure as described in Section 5.1. Certainly, the training sample was disjoint from the set of instances used in the main experiments. We did not include large instances (e.g., with $n \geqslant 300$) in the training dataset. This decision was motivated by two reasons. First, our comparison of the SA-VNS algorithm with state-of-the-art methods was limited to instances containing at most 200 activities. Second, incorporating large-scale instances into the parameter tuning of SA-VNS would require substantial computational resources. Our parameter sensitivity analysis was based on a simple procedure. We tested a number of predefined values of one parameter while fixing the other VNS parameters at reasonable values, which were chosen

during preliminary experimentation. For each parameter setting and each instance, we performed 10 independent runs of SA-VNS and computed the average objective function value over these runs. The time limit per run was set to 5 s for $n = 100$ and 20 s for $n = 200$. The experimental results show that VNS is robust in finding good solutions for a rather wide range of parameter values. Our first step in parameter tuning was to select a good value for η_2. We ran SA-VNS with $\eta_2 \in \{0.2, 0.3, 0.4, 0.5\}$ (note that η_2 cannot exceed 0.5). Good quality solutions were obtained for $\eta_2 \in \{0.3, 0.4, 0.5\}$. A slightly better performance was observed for $\eta_2 = 0.4$. SA-VNS with $\eta_2 = 0.4$ obtained the best average values for 7 (out of 10) instances. Results of this kind for other values of η_2 were worse: 0, 3, and 4 instances for $\eta_2 = 0.2$, $\eta_2 = 0.3$, and $\eta_2 = 0.5$, respectively. We thus fixed η_2 at 0.4. Next, we examined the following values of the parameter η_1: 0.01, 0.02, 0.05, 0.1, 0.2, 0.3, and 0.4 (we remind that $\eta_1 \leqslant \eta_2$). The results were incredibly similar for the η_1 values between 0.1 and 0.4. The algorithm with $\eta_1 < 0.1$ performed equally well for instances with $n = 100$ and marginally worse for instances with $n = 200$. We decided to fix η_1 at 0.2. Furthermore, we evaluated the effect of the parameter $r_{\min}$ on the performance of SA-VNS. We experimented with the following values for $r_{\min}$: 1, 5, 10, 20, and 50. The results have shown that SA-VNS is not sensitive to the values of $r_{\min}$ in the range 1 to 20. The performance of SA-VNS started to deteriorate at $r_{\min} = 50$. Based on these findings, we pretty arbitrarily set $r_{\min}$ to 5. In the next experiment, we assessed the influence of the parameter β on the quality of solutions. We ran SA-VNS with $\beta \in \{1, 2, 3, 4, 5, 10, 20, 50\}$. We found that the algorithm is fairly robust for β values in the range 3 to 20. The performance of SA-VNS decreased for $\beta = 2$ and $\beta = 50$. Its performance rapidly declined for $\beta = 1$. Based upon these results, we set β to 5, which is the middle element in the list 3, 4, 5, 10, and 20.

To finish configuring our SA and VNS hybrid, we needed to set the CPU quota parameter μ. This parameter serves for dividing the processor time resources between the SA and VNS algorithms (see Section 2). To choose a proper value of μ, we ran SA-VNS on the same set of instances as before. We varied μ from 0 to 1 in increments of 0.1. For each FLMP instance in the training set, the accuracy of solutions produced by SA-VNS was measured using the difference between the average objective function value over 10 runs and the value of the best solution obtained from all runs of SA-VNS for all values of μ. Let us denote by ΔF_{av} this difference averaged over all instances in the set. In Figure 3, ΔF_{av} is plotted as a function of μ. We note that the pure VNS algorithm is obtained by setting μ to 0, and the pure multi-start SA algorithm is activated by setting μ to 1. As shown in Figure 3, we see that the pure VNS is the least efficient SA-VNS configuration. However, the pure multi-start SA is not the best choice either. We observe that reasonable values of the parameter μ fall in the interval $[0.4, 0.7]$. With these results, we fixed μ to 0.5, thus giving 50% of the time limit to each of the two components of the hybrid algorithm.

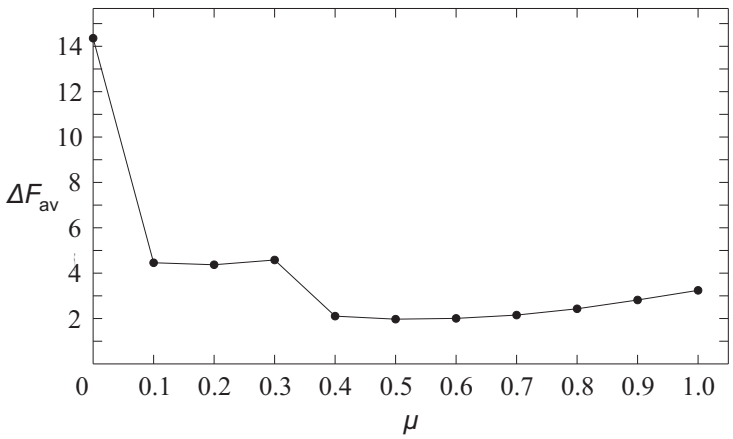


Figure 3. Effect of the hybridization of SA and VNS.

5.3. Numerical Results

We now provide computational comparisons between our hybrid heuristic and state-of-the-art algorithms for the FLMP. One such algorithm is the insertion-based simulated annealing algorithm of Lin et al. [29]. These authors use the name ISA to refer to this algorithm. We decided to keep this name. Lin et al. [29] also proposed an insertion-based genetic algorithm (IGA) for solving the FLMP. According to the computational experiments in [29], both algorithms, ISA and IGA, performed equally well. For comparison purposes, we implemented the ISA algorithm. During the implementation, we strictly followed the description provided in [29], particularly the pseudocode presented in Tables I and III of that paper. Both the SA-VNS and ISA algorithms were executed on the same hardware under single-threaded settings. We note that our SA-VNS algorithm and IGA can be indirectly compared by confronting the results of this section with those presented by Lin et al. [29]. The comparison also includes the imperialist competitive algorithm of Khanmirza et al. [30].

By running SA-VNS and ISA on the same hardware platform (Intel Core i5-6200U CPU), we ensured the consistency of the computational conditions. For large-scale problem instances, the ISA algorithm was observed to require substantially longer runtimes to converge to a reasonable solution. Therefore, while a fixed maximum CPU time was imposed on our method, ISA was allowed to run until its own termination criterion was met, which in many cases exceeded the imposed time limit. This choice was made to avoid penalizing ISA due to premature termination and to provide a conservative and fair comparison.

The results of solving the benchmark FLMP instances from the literature are reported in Table 3. They were obtained by running each of the three algorithms 30 times per instance. The second column of the table gives the number of activities. The next nine columns contain the results of the ICA, ISA, and SA-VNS algorithms. The value of the best solution out of 30 runs is denoted by F_{best}, and the average value of 30 solutions found by an algorithm is denoted by F_{av}. Columns labeled "Time" show the average running time (in seconds) to the best solution in a run of an algorithm. The results of the ICA (Columns 3–5 of Table 3) are taken from the paper of Khanmirza et al. [30]. These authors ran ICA on a computer with an Intel Core i7 (3.70 GHz) processor. (Intel Corporation, Santa Clara, CA, USA) However, they provided only an Intel Core brand modifier i7, and not the full processor name. Therefore, the comparison of SA-VNS and ICA is somewhat complicated. The single-thread rating of our PC (Intel Core i5-6200U, 2.30 GHz (Intel Corporation, Santa Clara, CA, USA)) is 1575 (for example, these data are available from https://www.cpubenchmark.net/singleCompare.php (accessed on 7 January 2026)). One might expect that the single-thread performance of the computer used by Khanmirza et al. [30] is comparable with, or better than, our laptop's CPU. If so, our system would have no speed advantage over the computer used in [30]. In any case, since an exact comparison is not possible, the reported running times are provided in Table 3 solely for reference.

We see from Table 3 that all the algorithms performed well on the benchmark problems. They easily achieved the best result in all runs for each problem instance (except ICA for `austin'96`, for which $F_{\text{av}} = 133.2$). It can also be observed that SA-VNS took much less time than ISA for the five larger instances. The overall conclusion from the experiment is that the benchmark FLMP instances were incredibly easy to solve.

Table 3. Comparison of the SA-VNS algorithm with the ICA of Khanmirza et al. [30] and ISA of Lin et al. [29].

Instance	n	ICA [30]			ISA [29]			SA-VNS		
		F_{best}	F_{av}	Time(s)	F_{best}	F_{av}	Time(s)	F_{best}	F_{av}	Time(s)
kusiak'91	12	6	6	<0.1	6	6	<0.1	6	6	<0.1
steward'81	20	24	24	0.3	24	24	0.4	24	24	<0.1
automobile	46	–	–	–	451	451	15.0	451	451	0.3
qian'14	48	–	–	–	106	106	10.5	106	106	0.4
austin'96	51	133	133.2	9.8	133	133	27.4	133	133	0.3
microprocessor	60	–	–	–	386	386	33.4	386	386	0.3
strategy	62	–	–	–	71	71	82.2	71	71	0.3

Tables 4 and 5 summarize the results of SA-VNS and ISA on the randomly generated problem instances of a size up to $n = 200$. The first column of the tables contains the names of the instances. The first number within the name denotes the number of activities. The second column shows the density of the input DSM. The F_{best} values in Table 4 are obtained from 10 independent runs of ISA (Column 3) and SA-VNS (Column 5). We also provide, for each algorithm, the success rate (denoted as SR), which is the fraction of times the algorithm outputs a solution of value F_{best} (Columns 4 and 6 in Table 4). Table 5 shows the F_{av} value, the standard deviation of the objective function values, σ, and the average running time (in seconds) to the best solution in a run for ISA (Columns 3–5) and for SA-VNS (Columns 6–8). Boldface is used to highlight those cases where SA-VNS obtained better F_{best} (Table 4) or better F_{av} (Table 5) values than ISA. We can observe from the tables that our algorithm demonstrates a good performance. Compared with ISA regarding F_{best}, SA-VNS outperformed in 13 cases and tied in the remaining 22 cases. Notably, SA-VNS achieved a perfect success rate of 10/10 for a larger number of instances than ISA (31 vs. 14). From Table 5, we can see that the dominance of SA-VNS is even more pronounced regarding the mean performance. It produced better average results than ISA for 21 problem instances, including all instances of size 100 and more. We notice that for $n = 30$ and the four instances with $n = 60$, both algorithms found the best solution in each of the 10 runs. For illustration purposes, as shown in Figure 4, we show the plots of the objective function value versus the run number for the two problem instances used in the experiment: `fp-150-2` and `fp-200-5`. As can be seen, SA-VNS consistently yielded better quality solutions than the ISA algorithm. The results in Table 5 also indicate that the SA-VNS algorithm outperformed ISA with generally smaller standard deviations of the objective values.

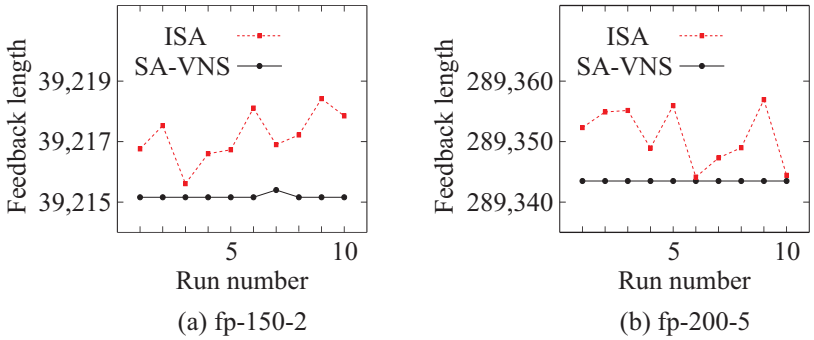


Figure 4. Feedback length versus run number.

Table 4. Comparison of SA-VNS with the state-of-the-art algorithm ISA from Lin et al. [29] on the FLMP instances with $n \leqslant 200$—best solution values (SR denotes the success rate).

Instance	Density (%)	ISA [29]		SA-VNS	
		F_{best}	SR	F_{best}	SR
fp-30-1	10	49.21	10/10	49.21	10/10
fp-30-2	10	40.36	10/10	40.36	10/10
fp-30-3	20	133.78	10/10	133.78	10/10
fp-30-4	20	169.23	10/10	169.23	10/10
fp-30-5	30	323.52	10/10	323.52	10/10
fp-30-6	30	306.51	10/10	306.51	10/10
fp-30-7	40	487.37	10/10	487.37	10/10
fp-30-8	40	484.75	10/10	484.75	10/10
fp-30-9	50	835.37	10/10	835.37	10/10
fp-30-10	50	687.07	10/10	687.07	10/10
fp-60-1	10	684.39	10/10	684.39	10/10
fp-60-2	10	569.65	8/10	569.65	10/10
fp-60-3	20	2134.64	8/10	2134.64	10/10
fp-60-4	20	1794.06	9/10	1794.06	10/10
fp-60-5	30	3126.51	10/10	3126.51	10/10
fp-60-6	30	3197.11	10/10	3197.11	10/10
fp-60-7	40	5381.64	8/10	5381.64	10/10
fp-60-8	40	5317.66	9/10	5317.66	10/10
fp-60-9	50	6734.01	10/10	6734.01	10/10
fp-60-10	50	6886.26	5/10	6886.26	10/10
fp-100-1	10	3795.32	6/10	3795.32	10/10
fp-100-2	20	10,379.05	4/10	10,379.05	10/10
fp-100-3	30	18,182.62	3/10	**18,182.38**	10/10
fp-100-4	40	26,072.53	1/10	**26,072.20**	10/10
fp-100-5	50	33,721.92	1/10	**33,721.90**	10/10
fp-150-1	10	16,192.93	1/10	**16,192.50**	8/10
fp-150-2	20	39,215.61	1/10	**39,215.16**	9/10
fp-150-3	30	63,833.26	1/10	**63,831.99**	10/10
fp-150-4	40	88,498.23	1/10	**88,498.16**	10/10
fp-150-5	50	118,012.66	1/10	**118,010.53**	10/10
fp-200-1	10	42,719.40	1/10	**42,718.98**	1/10
fp-200-2	20	98,504.32	1/10	**98,502.94**	10/10
fp-200-3	30	159,114.79	1/10	**159,109.90**	8/10
fp-200-4	40	227,074.66	1/10	**227,060.86**	10/10
fp-200-5	50	289,344.14	1/10	**289,343.49**	10/10

Comparing the running times, we observe the overwhelming superiority of SA-VNS. Figure 5 shows the convergence speed comparison between the two algorithms for a couple of problem instances: `fp-60-8` and `fp-100-1`. We see that SA-VNS identifies the best solution particularly quickly, whereas ISA needs a much longer time to do so. For instances with $n = 200$, the average time taken by ISA and SA-VNS is 26391.1 s and 91.4 s, respectively. The speed-up factor of 288 on this subset of instances was achieved.

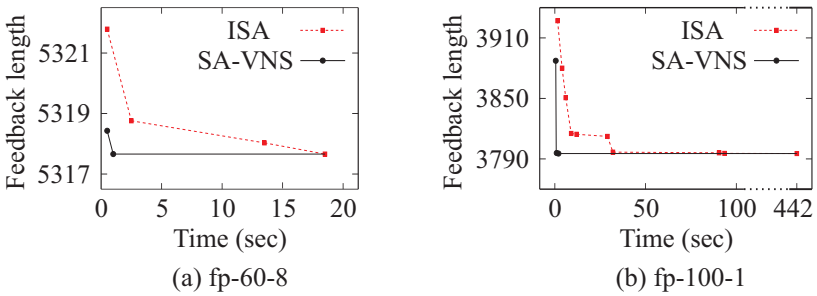


Figure 5. Convergence speed of SA-VNS and ISA.

Table 5. Comparison of SA-VNS with the state-of-the-art algorithm ISA from Lin et al. [29] on the FLMP instances with $n \leqslant 200$—average results (σ denotes the standard deviation).

Instance	Density (%)	ISA [29]			SA-VNS		
		F_{av}	σ	Time (s)	F_{av}	σ	Time (s)
fp-30-1	10	49.21	0	2.9	49.21	0	0.2
fp-30-2	10	40.36	0	2.3	40.36	0	0.1
fp-30-3	20	133.78	0	3.4	133.78	0	1.5
fp-30-4	20	169.23	0	2.2	169.23	0	0.2
fp-30-5	30	323.52	0	2.9	323.52	0	0.2
fp-30-6	30	306.51	0	2.6	306.51	0	0.3
fp-30-7	40	487.37	0	7.0	487.37	0	0.1
fp-30-8	40	484.75	0	4.8	484.75	0	0.1
fp-30-9	50	835.37	0	4.6	835.37	0	0.2
fp-30-10	50	687.07	0	3.5	687.07	0	0.1
fp-60-1	10	684.39	0	56.2	684.39	0	0.8
fp-60-2	10	569.69	0.08	37.1	**569.65**	0	0.8
fp-60-3	20	2134.67	0.08	58.4	**2134.64**	0	0.8
fp-60-4	20	1794.08	0.05	37.9	**1794.06**	0	0.6
fp-60-5	30	3126.51	0	46.4	3126.51	0	0.7
fp-60-6	30	3197.11	0	16.5	3197.11	0	0.6
fp-60-7	40	5381.65	0.02	25.8	**5381.64**	0	0.5
fp-60-8	40	5317.70	0.11	19.8	**5317.66**	0	0.9
fp-60-9	50	6734.01	0	31.8	6734.01	0	0.5
fp-60-10	50	6886.30	0.04	89.8	**6886.26**	0	0.7
fp-100-1	10	3795.38	0.07	742.6	**3795.32**	0	1.8
fp-100-2	20	10,379.14	0.16	1203.4	**10,379.05**	0	2.6
fp-100-3	30	18,182.79	0.22	863.8	**18,182.38**	0	1.7
fp-100-4	40	26,072.91	0.32	763.8	**26,072.20**	0	1.8
fp-100-5	50	33,722.48	0.32	1142.7	**33,721.90**	0	2.8
fp-150-1	10	16,194.01	0.59	6138.2	**16,192.60**	0.21	49.6
fp-150-2	20	39,217.16	0.80	5648.9	**39,215.18**	0.07	31.8
fp-150-3	30	63,836.63	2.22	8536.0	**63,831.99**	0	9.8
fp-150-4	40	88,499.39	0.87	6119.7	**88,498.16**	0	11.7
fp-150-5	50	118,014.57	1.40	5153.8	**118,010.53**	0	5.1
fp-200-1	10	42,720.96	1.23	34,197.0	**42,720.37**	1.29	92.7
fp-200-2	20	98,504.97	1.31	29,276.7	**98,502.94**	0	31.3
fp-200-3	30	159,117.88	3.36	21,657.1	**159,109.91**	0.02	127.0
fp-200-4	40	227,078.84	3.93	22,753.3	**227,060.86**	0	120.5
fp-200-5	50	289,350.91	4.54	24,071.2	**289,343.49**	0	85.6

When the size of the FLMP increases beyond 200 activities, the execution time of the ISA algorithm becomes unreasonably long. Therefore, we did not try to include ISA in our experiments with problem instances of a size greater than 200. Instead, for the sake of comparison, we ran SA and VNS algorithms on these instances. The results are presented in Tables 6 and 7. Their first column identifies the problem instance. The labels of the columns for each algorithm (SA, VNS, and SA-VNS) have the same meaning as those for the algorithms in Tables 4 and 5. The bottom row of each table shows the results averaged over all 25 instances. The best value of F_{best} in Table 6 and F_{av} in Table 7 for each instance is highlighted in bold. From the tables, we can see that SA-VNS outperformed the other two algorithms. Both SA-VNS and VNS were able to produce the best solution for 12 problem instances. However, SA-VNS achieved the best average result for 20 instances, whereas VNS performed this for one instance only (as highlighted in Table 7). Inspecting Table 6, one can see that the success rate of the algorithms for most instances (in the case of SA, for all of them) was 1. This suggests that the FLMP instances containing 250 or more activities were quite difficult for the algorithms involved in our experiments. We also observe that the SA algorithm found the best solution and the best average cost solution for 2 and 4 instances, respectively. It can be concluded that both SA-VNS and SA have the ability to obtain better average results than VNS. Moreover, SA and SA-VNS exhibited much smaller standard deviation values compared with those obtained from VNS. Another observation is that all three algorithms are comparable with respect to computation time. This is illustrated by the

plots in Figure 6, which show the average computation time as a function of the number of activities. For larger problem instances, the VNS algorithm is only slightly slower than SA and SA-VNS. Overall, we can conclude from the tables that the combination of the variable neighborhood search method with the simulated annealing technique resulted in a promising framework for solving the FLMP.

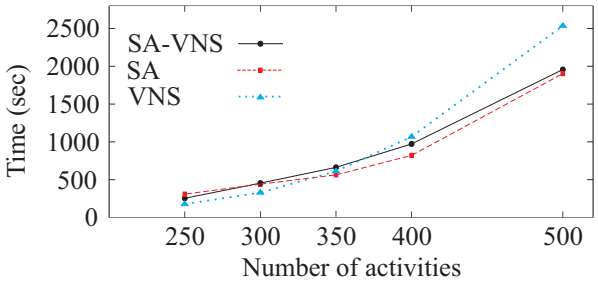


Figure 6. Computation time between SA-VNS, SA, and VNS comparison.

Table 6. Best results obtained by the SA-VNS, SA, and VNS algorithms on the FLMP instances with $n \in \{250, 300, 350, 400, 500\}$ (SR denotes the success rate).

Instance	Density (%)	SA		VNS		SA-VNS	
		F_{best}	SR	F_{best}	SR	F_{best}	SR
fp-250-1	10	83,963.14	1/10	**83,961.92**	1/10	83,962.00	1/10
fp-250-2	20	199,922.69	1/10	**199,921.97**	3/10	199,922.61	2/10
fp-250-3	30	320,662.68	1/10	320,678.30	1/10	**320,662.56**	5/10
fp-250-4	40	441,299.40	1/10	**441,298.62**	1/10	441,299.17	1/10
fp-250-5	50	563,247.92	1/10	**563,247.65**	2/10	**563,247.65**	6/10
fp-300-1	10	**154,330.57**	1/10	154,331.09	1/10	154,331.39	1/10
fp-300-2	20	356,043.84	1/10	**356,039.22**	1/10	356,040.62	1/10
fp-300-3	30	565,490.48	1/10	**565,489.45**	1/10	565,489.63	1/10
fp-300-4	40	783,711.26	1/10	**783,709.32**	3/10	783,709.91	1/10
fp-300-5	50	1,000,625.73	1/10	**1,000,625.45**	2/10	1,000,625.49	1/10
fp-350-1	10	249,112.84	1/10	249,114.28	1/10	**249,110.44**	1/10
fp-350-2	20	562,954.23	1/10	**562,951.61**	3/10	562,952.56	1/10
fp-350-3	30	911,284.92	1/10	**911,282.29**	1/10	911,282.43	1/10
fp-350-4	40	1,231,392.42	1/10	**1,231,391.95**	9/10	1,231,392.08	3/10
fp-350-5	50	1,613,558.41	1/10	1,613,566.90	1/10	**1,613,556.83**	1/10
fp-400-1	10	**386,653.47**	1/10	386,743.34	1/10	386,654.21	1/10
fp-400-2	20	876,251.03	1/10	876,397.39	1/10	**876,249.84**	1/10
fp-400-3	30	1,364,351.72	1/10	**1,364,350.19**	1/10	1,364,351.24	1/10
fp-400-4	40	1,891,287.22	1/10	1,891,288.12	1/10	**1,891,286.00**	1/10
fp-400-5	50	2,402,178.53	1/10	2,402,201.11	1/10	**2,402,177.33**	1/10
fp-500-1	10	786,842.68	1/10	786,969.73	1/10	**786,842.67**	1/10
fp-500-2	20	1,758,886.55	1/10	1,759,030.41	1/10	**1,758,877.46**	1/10
fp-500-3	30	2,761,373.74	1/10	2,761,365.07	1/10	**2,761,360.16**	1/10
fp-500-4	40	3,743,210.09	1/10	3,743,215.54	1/10	**3,743,207.96**	1/10
fp-500-5	50	4,769,878.64	1/10	4,769,890.27	1/10	**4,769,865.88**	1/10
Average		1,191,140.57	1/10	1,191,162.45	1/10 1.64/10	1,191,138.32	1.48/10

To ensure a more accurate assessment of the results, we applied the Wilcoxon signed-rank test for each pair of algorithms. The pairwise comparison results are summarized in Table 8. The column labeled OFV indicates which objective function values were compared: best solution values (F_{best} in Table 6) in the first row for each pair of algorithms and average solution values (F_{av} in Table 7) in the second row. The next three columns report the number of instances on which the first algorithm in the pair found a better (#wins), an equally good (#ties), or an inferior (#losses) solution when compared with the second algorithm. The penultimate column of the table shows the p-values obtained using the Wilcoxon test. For the last column, we used a standard significance level of 0.05 to declare whether a significant difference existed or not. The value "Yes" means that the results of the first

algorithm were better than those of the second algorithm, while the value "No" means that there was no statistical difference between the results of the two algorithms. A quick inspection of Table 8 shows that our SA-VNS hybrid technique was superior to the SA and VNS algorithms. The latter, in particular, exhibited poor performance regarding the average objective function value. These conclusions echo the findings in Section 5.2 (see Figure 3). We also applied the Friedman test followed by the Nemenyi post hoc test at a significance level of 0.05. The computed Friedman statistics were 9.26 and 38.32 for the results reported in Tables 6 and 7, respectively, both of which exceeded the critical value of 6.08. Therefore, we conclude that the objective values obtained by the compared algorithms were statistically significantly different. The Nemenyi post hoc test further confirmed statistically significant pairwise differences between SA, VNS, and SA-VNS. Specifically, the ranking from best to worst was SA-VNS, VNS, and SA for the best objective values, and SA-VNS, SA, and VNS for the average objective values.

Table 7. Average results of the SA-VNS, SA, and VNS algorithms on the FLMP instances with $n \in \{250, 300, 350, 400, 500\}$ (σ denotes the standard deviation).

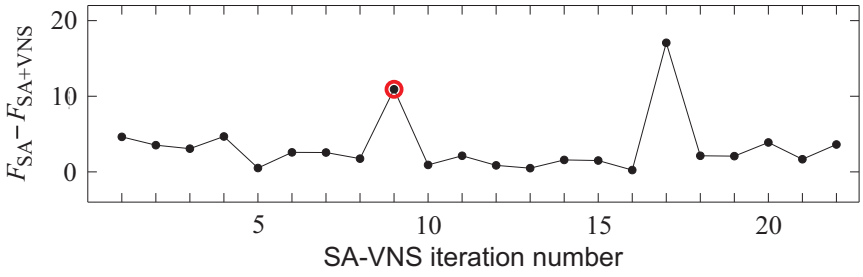
Instance	SA			VNS			SA-VNS		
	F_{av}	σ	Time (s)	F_{av}	σ	Time (s)	F_{av}	σ	Time (s)
fp-250-1	**83,972.41**	9.69	309.1	83,989.08	24.31	136.0	83,974.56	9.74	244.9
fp-250-2	199,923.74	0.75	311.8	199,928.60	9.58	213.0	**199,922.82**	0.18	225.5
fp-250-3	320,663.10	0.29	262.2	320,708.82	15.33	168.2	**320,662.58**	0.02	294.8
fp-250-4	**441,304.60**	3.96	346.7	441,348.15	39.02	151.5	441,305.04	4.68	273.5
fp-250-5	563,248.58	0.55	313.9	563,256.91	17.61	265.7	**563,247.91**	0.43	232.8
fp-300-1	154,334.36	1.75	415.5	154,371.06	39.30	372.6	**154,332.71**	0.84	448.9
fp-300-2	356,050.91	4.20	447.8	356,096.58	30.29	263.2	**356,049.30**	8.48	448.3
fp-300-3	565,494.19	3.64	450.4	565,562.22	78.81	346.5	**565,491.61**	3.70	494.3
fp-300-4	**783,712.53**	1.35	519.8	783,736.29	35.46	400.7	783,719.28	11.83	501.1
fp-300-5	1,000,626.47	0.60	372.3	1,000,664.64	22.00	314.6	**1,000,625.63**	0.08	379.9
fp-350-1	249,116.40	2.01	628.0	249,159.59	61.47	756.2	**249,114.50**	2.24	845.7
fp-350-2	562,956.08	1.10	646.9	**562,952.49**	0.87	788.0	562,953.91	0.83	516.4
fp-350-3	911,290.26	3.48	511.5	911,467.91	85.92	619.8	**911,286.78**	5.07	690.4
fp-350-4	1,231,394.27	1.76	402.5	1,231,414.31	67.08	302.1	**1,231,392.97**	1.83	565.7
fp-350-5	1,613,563.93	3.35	629.8	1,613,713.99	176.64	643.8	**1,613,560.33**	2.77	696.6
fp-400-1	386,658.48	3.23	695.9	386,856.75	93.29	1196.6	**386,658.05**	2.89	1092.5
fp-400-2	876,254.97	2.70	894.4	876,564.70	164.65	1060.9	**876,253.50**	1.92	916.2
fp-400-3	1,364,357.17	2.89	841.7	1,364,492.14	110.10	1138.9	**1,364,356.55**	2.77	941.0
fp-400-4	1,891,293.04	4.25	1046.7	1,891,422.22	91.23	847.8	**1,891,292.63**	8.05	1063.2
fp-400-5	2,402,186.23	4.66	629.5	2,402,373.64	207.91	1147.8	**2,402,186.21**	5.89	852.6
fp-500-1	786,849.44	3.38	1564.4	787,245.14	185.07	2827.8	**786,848.76**	4.09	2209.5
fp-500-2	1,758,894.28	6.16	2356.0	1,759,277.74	136.88	2164.1	**1,758,892.14**	6.15	1499.0
fp-500-3	**2,761,399.54**	19.64	1774.5	2,761,692.28	249.29	3014.6	2,761,401.62	23.86	2320.4
fp-500-4	3,743,219.78	6.21	1772.8	3,744,446.32	1144.43	2624.6	**3,743,218.69**	7.62	1748.6
fp-500-5	4,769,888.93	6.08	2048.1	4,770,378.26	420.46	2095.6	**4,769,888.73**	11.21	2007.1
Average	1,191,146.15	3.91	805.7	1,191,324.79	140.28	954.4	1,191,145.47	5.09	860.3

Table 8. Summary of the results obtained by the tested algorithms for large-scale instances.

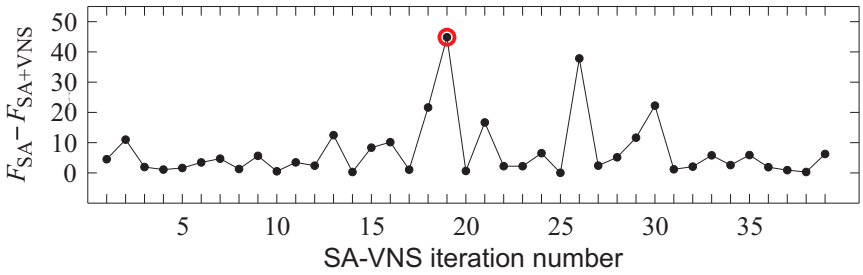
Algorithm Pair	OFV	#Wins	#Ties	#Losses	*p*-Value	Statistical Significance
SA-VNS vs. SA	Best	23	0	2	<0.001	Yes
	Average	21	0	4	0.01	Yes
SA-VNS vs. VNS	Best	12	1	12	0.05	Yes
	Average	24	0	1	<0.001	Yes
SA vs. VNS	Best	12	0	13	>0.2	No
	Average	24	0	1	<0.001	Yes

As shown in Figure 7, we illustrated the performance of SA-VNS on a couple of FLMP instances, `fp-200-1` and `fp-500-2`, from our testbed. In each iteration of SA-VNS, the SA algorithm was first applied. Let us assume that the objective function value achieved by SA is denoted by F_{SA}. In the second stage of the iteration, the SA solution can be improved

by executing the VNS algorithm. We denote the objective function value of the resulting solution by $F_{\mathrm{SA+VNS}}$ (thus, the objective function value achieved by SA-VNS equals the minimum of $F_{\mathrm{SA+VNS}}$ when it has taken over all SA-VNS iterations). Figure 7 depicts the plots of $F_{\mathrm{SA}} - F_{\mathrm{SA+VNS}}$ versus the SA-VNS iteration number. The average values of $F_{\mathrm{SA}} - F_{\mathrm{SA+VNS}}$ are 3.29 and 7.05 for `fp-200-1` and `fp-500-2`, respectively. They are calculated over 22 SA-VNS iterations for `fp-200-1` and 39 SA-VNS iterations for `fp-500-2`. The SA-VNS algorithm found the best solution for `fp-200-1` in Iteration No. 9 and the best solution for `fp-500-2` in Iteration No. 19. The corresponding values of $F_{\mathrm{SA}} - F_{\mathrm{SA+VNS}}$ in the figure are designated by a dot enclosed in a circle.



(a) fp-200-1



(b) fp-500-2

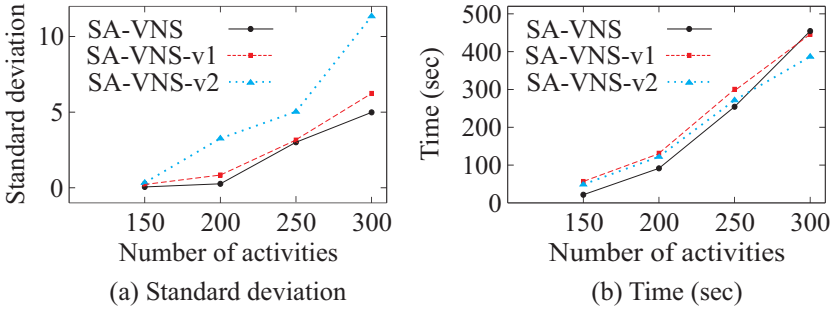
Figure 7. Decrease in the feedback length after applying VNS to a solution produced by SA.

The two key components of the VNS algorithm are the shaking and local search mechanisms. To assess the contribution of each component, we conducted an additional experiment. In one variant of the SA-VNS algorithm, the shaking procedure was removed; instead, each VNS iteration started from a randomly generated permutation (we refer to this variant as SA-VNS-v1). In another variant, we replaced our fast local search technique (described in Algorithm 8) with a straightforward local search implementation, in which the objective function values of the examined solutions were computed directly using Equation (1). In this version, denoted SA-VNS-v2, the shaking mechanism was retained.

We compared the original algorithm and its two variants on a set of FLMP instances containing between 150 and 300 activities. The experimental results are reported in Table 9 and Figure 8. The results indicate that SA-VNS-v1 and SA-VNS-v2 achieved the same F_{best} values as SA-VNS for only 7 and 6 out of 20 instances, respectively; for the remaining instances, they failed to match the performance of SA-VNS. An examination of the F_{av} values shows that SA-VNS consistently outperformed SA-VNS-v1 on average, while SA-VNS-v1, in turn, outperformed SA-VNS-v2 across all problem instances in the dataset. Part (a) of Figure 8 shows that SA-VNS-v2 exhibited a larger standard deviation of objective function values compared with SA-VNS and SA-VNS-v1, whereas the latter two algorithms are comparable regarding this metric. As shown in part (b) of Figure 8, all three algorithms required very similar computation times. Overall, these results demonstrate that SA-VNS significantly outperforms its variants SA-VNS-v1 and SA-VNS-v2.

Table 9. Performance comparison of SA-VNS and its two versions SA-VNS-v1 and SA-VNS-v2.

Instance	SA-VNS-v1		SA-VNS-v2		SA-VNS	
	F_{best}	F_{av}	F_{best}	F_{av}	F_{best}	F_{av}
fp-150-1	16,192.51	16,193.12	16,193.03	16,193.81	**16,192.50**	16,192.60
fp-150-2	**39,215.16**	39,215.69	**39,215.16**	39,215.92	**39,215.16**	39,215.18
fp-150-3	**63,831.99**	63,832.45	63,832.32	63,833.16	**63,831.99**	63,831.99
fp-150-4	**88,498.16**	88,498.19	**88,498.16**	88,498.21	**88,498.16**	88,498.16
fp-150-5	**118,010.53**	118,010.54	**118,010.53**	118,010.61	**118,010.53**	118,010.53
fp-200-1	42,719.48	42,721.48	42,719.48	42,723.03	**42,718.98**	42,720.37
fp-200-2	**98,502.94**	98,503.06	**98,502.94**	98,503.14	**98,502.94**	98,502.94
fp-200-3	**159,109.90**	159,110.33	**159,109.90**	159,111.40	**159,109.90**	159,109.91
fp-200-4	**227,060.86**	227,061.23	**227,060.86**	227,065.34	**227,060.86**	227,060.86
fp-200-5	289,343.51	289,345.31	289,343.51	289,353.14	**289,343.49**	289,343.49
fp-250-1	83,963.22	83,976.66	83,963.22	83,980.23	**83,962.00**	83,974.56
fp-250-2	199,923.34	199,924.11	199,923.34	199,926.26	**199,922.61**	199,922.82
fp-250-3	320,662.68	320,663.38	320,662.68	320,665.49	**320,662.56**	320,662.58
fp-250-4	441,300.59	441,306.46	441,308.64	441,318.96	**441,299.17**	441,305.04
fp-250-5	563,247.97	563,248.73	563,248.50	563,250.00	**563,247.65**	563,247.91
fp-300-1	154,333.13	154,334.78	154,334.05	154,342.15	**154,331.39**	154,332.71
fp-300-2	356,043.84	356,054.91	356,046.64	356,067.14	**356,040.62**	356,049.30
fp-300-3	565,490.48	565,496.49	565,493.30	565,513.26	**565,489.63**	565,491.61
fp-300-4	783,711.26	783,723.82	783,711.26	783,750.34	**783,709.91**	783,719.28
fp-300-5	1,000,625.73	1,000,626.95	1,000,626.81	1,000,632.56	**1,000,625.49**	1,000,625.63



(a) Standard deviation (b) Time (sec)

Figure 8. Comparison of SA-VNS with SA-VNS-v1 and SA-VNS-v2: (**a**) standard deviation; (**b**) computation time (in seconds).

5.4. Local Search Experiments

The performance of both our VNS heuristic and the ISA algorithm of Lin et al. [29] heavily depends on the local search technique used. To evaluate the convergence of the local search components of these algorithms, we conducted an experiment on a subset of random FLMP instances of various sizes. The local search procedure in ISA is called insertion-based heuristic [29]. We use the abbreviation IBH to refer to this heuristic. We ran both IBH and our LS procedure (shown in Algorithm 8) 100 times on each instance of size $n \leqslant 100$ and 10 times on each instance of size $n > 100$. The results are shown in Table 10. In each run, the algorithms started from a random initial solution. The efficiency of algorithms is measured regarding the average running time. We did not take into account the quality of the local minima the algorithm arrived at. We note that the IBH and LS algorithms were quite different from each other. Therefore, in general, they produced different solutions. However, the average objective function values obtained by these algorithms were very similar.

The results of the tested local search procedures were compared, as shown in Table 10. The problem instances selected for the experiment are listed in the first column. The second column shows the number of independent runs of the algorithms. The next two columns report the running time averaged over all runs for IBH and LS, respectively. The last column gives the ratio of the running times of IBH and LS. The table clearly shows that LS reached a local minimum much faster than IBH. Essentially, we see that, from the last column, LS took two orders of magnitude less computational time than the IBH procedure. Lin et al. [29] embedded IBH into their ISA algorithm. Based on this fact and the results of our experiment, one could argue that the slowness of IBH is the main reason why the

ISA algorithm takes a much greater computation time in comparison with our SA-VNS hybrid approach.

Table 10. Time comparison between LS and the local search algorithm of Lin et al. [29] (in seconds).

Instance	#Number of Runs	IBH [29]	LS	Time Ratio
fp-30-1	100	0.2	<0.1	12.7
fp-60-1	100	4.3	0.2	21.2
fp-100-1	100	44.3	1.1	40.6
fp-150-1	10	27.8	0.4	66.0
fp-200-1	10	107.5	1.2	89.3
fp-250-1	10	273.1	2.7	102.3
fp-300-1	10	658.7	5.2	126.2
fp-350-1	10	1222.6	9.0	136.6
fp-400-1	10	2157.1	13.7	157.9
fp-500-1	10	5743.0	33.4	172.2

6. Discussion

The design structure matrix has become an important modeling tool for managing complexity in product development projects. By representing process activities and their dependencies in a compact, square matrix, the DSM enables engineers and project managers to visualize the structure of information flows and to identify opportunities for process improvement. Its strength lies in highlighting feedback loops, i.e., iterations in which upstream activities require information from downstream activities. Such loops often disproportionately contribute to schedule delays and increased development costs. Two optimization problems arising from DSM analysis, the feedback minimization problem and the feedback length minimization problem (as given by Equation (2)), aim to reorder activities (or tasks) to reduce the impact of such iterative cycles. The FMP focuses on minimizing the number of feedback marks above the main diagonal of the DSM. This is useful for identifying activity sequences that achieve a more acyclic or feed-forward structure. However, minimizing the count of feedback alone may not fully capture the practical consequences of dealing with feedback loops. A single long-range feedback that spans many activities can be far more disruptive than several short-range ones. This limitation motivates the FLMP, which extends the feedback minimization problem by incorporating the length of feedback into the objective function. Feedback length corresponds to the distance between dependent activities in the chosen sequencing, which more directly reflects the expected duration and cost implications of rework or information iteration. As a result, the FLMP provides a more realistic basis for optimizing activity order in PD projects, especially in complex or tightly coupled engineering environments. By focusing on the reduction in long feedback loops, the FLMP aligns more closely with project management goals, such as shortening development time, reducing risk, and improving overall process efficiency. In practice, DSM-based optimization, using algorithms for the FLMP, supports decision makers in reorganizing processes to reduce iteration and uncertainty.

However, the FLMP is combinatorial and NP-hard, requiring heuristic or metaheuristic solution strategies for realistic PD project sizes. In this paper, we propose a hybrid approach that combines the strengths of the SA and VNS metaheuristics. The resulting SA-VNS algorithm compared very favorably with existing methods. Several factors contributed to its superior performance. First, the algorithm repeatedly applied both SA and VNS components, enabling effective exploration of the solution space. Second, SA produced high-quality initial solutions for the VNS phase, allowing VNS to focus immediately on the promising regions of the solution space. Third, VNS incorporated a fast local search procedure that significantly enhanced the algorithm's exploitation capabilities.

The proposed method, however, has several limitations. First, SA is not a fast-converging metaheuristic; therefore, SA-VNS may require long computation times for particularly large-scale FLMP instances. Second, SA-VNS is a single-solution algorithm and does not use a population of individuals. In some cases, algorithms employing evolutionary strategies may achieve better performance.

7. Concluding Remarks

In this paper, we implemented an idea to integrate the two metaheuristics, simulated annealing and variable neighborhood search, into a single approach for solving the FLMP. During a run of the hybrid algorithm, its components SA and VNS were executed iteratively. At each iteration, starting from a randomly generated initial permutation, SA produced an improved solution and submitted it to VNS for possible further improvement. The key point of the VNS algorithm is the use of a fast insertion neighborhood exploration procedure. The computational complexity of this procedure is proportional to the size of the neighborhood and, thus, is the smallest possible up to a constant factor.

We conducted computational experiments to compare the performance of our SA-VNS hybrid and the insertion-based simulated annealing heuristic, ISA, which is the state-of-the-art algorithm for the FLMP. The experimental results demonstrate a clear superiority of our approach over ISA. On the one hand, SA-VNS required two orders of magnitude less CPU time than the ISA algorithm. On the other hand, SA-VNS produced equally good or better results across all tested problem instances. In particular, SA-VNS was able to find better solutions than ISA on all instances of size 150 or more. Our secondary aim was to compare the SA-VNS hybrid technique against the SA and VNS heuristics when treated as stand-alone algorithms. Using rigorous statistical tests, our results show that SA-VNS is superior to each of SA and VNS when running alone. We also carried out an experiment to assess the performance of local search procedures employed in the SA-VNS and ISA algorithms. From the results, it can be concluded that our LS procedure is particularly fast and well suited for implementation in the VNS algorithm. Despite the good performance of the SA-VNS algorithm, several limitations can be identified. First, as the problem size increases, the SA component struggles to obtain high-quality solutions within a short time, which may negatively affect the overall performance of the algorithm. Second, the VNS component may fail to efficiently explore the solution space for large-scale problem instances. These observations are supported by the experimental results, which show relatively large standard deviation values for the VNS method. Overall, the performance of SA-VNS deteriorates for FLMP instances with $n \geqslant 250$. This can be seen in Table 6, where the success rate was the lowest possible, i.e., 1/10, in the majority of cases.

Future work may involve the development of reasonably fast metaheuristic-based algorithms capable of obtaining high-quality solutions for very large instances of the FLMP (say, with thousands of activities). An intriguing direction would be to apply evolutionary computing techniques (such as memetic algorithms) for solving the FLMP. Evolutionary algorithms can use the proposed local search procedure as a suitable mechanism for search intensification. Another important avenue for future work is to investigate the applicability of an SA and VNS hybrid algorithm for solving other optimization problems defined on permutations. In this line, it would be interesting to consider other DSM sequencing problems arising in the PD field. Important recent examples of such problems are scheduling interrelated activities in complex projects under high-order rework [71] and task sequencing to minimize project duration [72].

Author Contributions: Conceptualization, G.P., A.M. and Z.D.; methodology, G.P., A.M. and Z.D.; software, G.P. and A.M.; validation, Z.D.; formal analysis, Z.D.; investigation, G.P.; writing—original draft, G.P. and A.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The raw data supporting the conclusions of this article will be made available by the authors on request.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Abdelsalam, H.M.E.; Bao, H.P. A simulation-based optimization framework for product development cycle time reduction. *IEEE Trans. Eng. Manag.* **2006**, *53*, 69–85.
2. Meier, C.; Yassine, A.A.; Browning, T.R. Design process sequencing with competent genetic algorithms. *J. Mech. Des.* **2007**, *129*, 566–585.
3. Jun, H.B.; Suh, H.W. A modeling framework for product development process considering its characteristics. *IEEE Trans. Eng. Manag.* **2008**, *55*, 103–119.
4. Eppinger, S.D.; Browning, T.R. *Design Structure Matrix Methods and Applications*; The MIT Press: Cambridge, MA, USA, 2012.
5. Attari-Shendi, M.; Saidi-Mehrabad, M.; Gheidar-Kheljani, J. A comprehensive mathematical model for sequencing interrelated activities in complex product development projects. *IEEE Trans. Eng. Manag.* **2022**, *69*, 2619–2633.
6. Steward, D.V. *Systems Analysis and Management: Structure, Strategy and Design*; Petrocelli Books: New York, NY, USA, 1981.
7. Eppinger, S.D.; Whitney, D.E.; Smith, R.P.; Gebala, D.A. A model-based method for organizing tasks in product development. *Res. Eng. Des.* **1994**, *6*, 1–13.
8. Tang, D.; Zheng, L.; Li, Z.; Li, D.; Zhang, S. Re-engineering of the design process for concurrent engineering. *Comput. Ind. Eng.* **2000**, *38*, 479–491.
9. Browning, T.R.; Eppinger, S.D. Modeling impacts of process architecture on cost and schedule risk in product development. *IEEE Trans. Eng. Manag.* **2002**, *49*, 428–442.
10. Cho, S.H.; Eppinger, S.D. A simulation-based process model for managing complex design projects. *IEEE Trans. Eng. Manag.* **2005**, *52*, 316–328.
11. Lin, J.; Chai, K.H.; Wong, Y.S.; Brombacher, A.C. A dynamic model for managing overlapped iterative product development. *Eur. J. Oper. Res.* **2008**, *185*, 378–392.
12. Shaja, A.S.; Sudhakar, K. Optimized sequencing of analysis components in multidisciplinary systems. *Res. Eng. Des.* **2010**, *21*, 173–187.
13. Tripathy, A.; Eppinger, S.D. Structuring work distribution for global product development organizations. *Prod. Oper. Manag.* **2013**, *22*, 1557–1575.
14. Yang, Q.; Yao, T.; Lu, T.; Zhang, B. An overlapping-based design structure matrix for measuring interaction strength and clustering analysis in product development project. *IEEE Trans. Eng. Manag.* **2014**, *61*, 159–170.
15. Liu, A.; Hu, H.; Zhang, X.; Lei, D. Novel two-phase approach for process optimization of customer collaborative design based on fuzzy-QFD and DSM. *IEEE Trans. Eng. Manag.* **2017**, *64*, 193–207.
16. Sun, Q.C.; Huang, W.Q.; Jiang, Y.J.; Sun, W. Information flow scheduling in concurrent multi-product development based on DSM. *Chin. J. Mech. Eng.* **2017**, *30*, 1101–1111.
17. Browning, T.R. Applying the design structure matrix to system decomposition and integration problems: A review and new directions. *IEEE Trans. Eng. Manag.* **2001**, *48*, 292–306.
18. Karniel, A.; Reich, Y. From DSM-based planning to design process simulation: A review of process scheme logic verification issues. *IEEE Trans. Eng. Manag.* **2009**, *56*, 636–649.
19. Browning, T.R. Design structure matrix extensions and innovations: A survey and new opportunities. *IEEE Trans. Eng. Manag.* **2016**, *63*, 27–52.
20. Piccirillo, I.N.; De Almeida, L.F.M.; De Araújo Júnior, L.Q.; Da Silva, S.L. Design structure matrix and project management: Bibliometric analysis. *Product* **2017**, *15*, 86–91.
21. Ahmadi, R.; Roemer, T.A.; Wang, R.H. Structuring product development processes. *Eur. J. Oper. Res.* **2001**, *130*, 539–558.
22. Qian, Y.; Lin, J.; Goh, T.N.; Xie, M. A novel approach to DSM-based activity sequencing problem. *IEEE Trans. Eng. Manag.* **2011**, *58*, 688–705.
23. Gebala, D.A.; Eppinger, S.D. Methods for analyzing design procedures. In Proceedings of the ASME 3rd International Conference on Design Theory and Methodology, Miami, Florida, USA, 22–25 September 1991; The American Society of Mechanical Engineers: New York, NY, USA, 1991; pp. 227–233. https://doi.org/10.1115/DETC1991-0052.
24. Lancaster, J.; Cheng, K. A fitness differential adaptive parameter controlled evolutionary algorithm with application to the design structure matrix. *Int. J. Prod. Res.* **2008**, *46*, 5043–5057.
25. Qian, Y.; Lin, J. Organizing interrelated activities in complex product development. *IEEE Trans. Eng. Manag.* **2014**, *61*, 298–309.

26. Shang, Z.; Zhao, S.; Qian, Y.; Lin, J. Exact algorithms for the feedback length minimisation problem. *Int. J. Prod. Res.* **2019**, *57*, 544–559.
27. Altus, S.S.; Kroo, I.M.; Gage, P.J. A genetic algorithm for scheduling and decomposition of multidisciplinary design problems. *J. Mech. Des.* **1996**, *118*, 486–489.
28. Whitfield, R.I.; Duffy, A.H.B.; Coates, G.; Hills, W. Efficient process optimization. *Concurr. Eng. Res. Appl.* **2003**, *11*, 83–92.
29. Lin, J.; Huang, W.; Qian, Y.; Zhao, X. Scheduling interrelated activities using insertion-based heuristics. *IEEE Trans. Eng. Manag.* **2018**, *65*, 113–127.
30. Khanmirza, E.; Haghbeigi, M.; Yazdanjue, N. Enhanced genetic and imperialist competitive based algorithms for reducing design feedbacks in the design structure matrix. *IEEE Trans. Eng. Manag.* **2023**, *70*, 3156–3170.
31. Yazdanjue, N.; Khanmirza, E. An application-specific approach for design structure matrix optimization: Focusing on the cross application of modularization and sequencing methods. *IEEE Trans. Eng. Manag.* **2023**, *70*, 2093–2114.
32. Mladenović, N.; Todosijević, R.; Urošević, D.; Ratli, M. Solving the capacitated dispersion problem with variable neighborhood search approaches: From basic to skewed VNS. *Comput. Oper. Res.* **2022**, *139*, 105622.
33. Matijević, L.; Đurasević, M.; Jakobović, D. A variable neighborhood search method with a tabu list and local search for optimizing routing in trucks in maritime ports. *Mathematics* **2023**, *11*, 3740.
34. Herrán, A.; Colmenar, J.M.; Mladenović, N.; Duarte, A. A general variable neighborhood search approach for the minimum load coloring problem. *Optim. Lett.* **2023**, *17*, 2065–2086.
35. Lu, S.; Pei, J.; Liu, X.; Qian, X.; Mladenović, N.; Pardalos, P.M. Less is more: Variable neighborhood search for integrated production and assembly in smart manufacturing. *J. Sched.* **2020**, *23*, 649–664.
36. Mladenović, N.; Alkandari, A.; Pei, J.; Todosijević, R.; Pardalos, P.M. Less is more approach: Basic variable neighborhood search for the obnoxious p-median problem. *Int. Trans. Oper. Res.* **2020**, *27*, 480–493.
37. Yilmaz, Y.; Kalayci, C.B. Variable neighborhood search algorithms to solve the electric vehicle routing problem with simultaneous pickup and delivery. *Mathematics* **2022**, *10*, 3108.
38. Sánchez-Oro, J.; Mladenović, N.; Duarte, A. General variable neighborhood search for computing graph separators. *Optim. Lett.* **2017**, *11*, 1069–1089.
39. Todosijević, R.; Hanafi, S.; Urošević, D.; Jarboui, B.; Gendron, B. A general variable neighborhood search for the swap-body vehicle routing problem. *Comput. Oper. Res.* **2017**, *78*, 468–479.
40. Todosijević, R.; Benmansour, R.; Hanafi, S.; Mladenović, N.; Artiba, A. Nested general variable neighborhood search for the periodic maintenance problem. *Eur. J. Oper. Res.* **2016**, *252*, 385–396.
41. Salhi, S.; Imran, A.; Wassan, N.A. The multi-depot vehicle routing problem with heterogeneous vehicle fleet: Formulation and a variable neighborhood search implementation. *Comput. Oper. Res.* **2014**, *52*, 315–325.
42. Krimi, I.; Benmansour, R.; Todosijević, R.; Mladenović, N.; Ratli, M. A no-delay single machine scheduling problem to minimize total weighted early and late work. *Optim. Lett.* **2023**, *17*, 2113–2131.
43. Mladenović, N.; Urošević, D.; Hanafi, S.; Ilić, A. A general variable neighborhood search for the one-commodity pickup-and-delivery travelling salesman problem. *Eur. J. Oper. Res.* **2012**, *220*, 270–285.
44. Krimi, I.; Todosijević, R.; Benmansour, R.; Ratli, M.; El Cadi, A.A.; Aloullal, A. Modelling and solving the multi-quays berth allocation and crane assignment problem with availability constraints. *J. Glob. Optim.* **2020**, *78*, 349–373.
45. Mladenović, N.; Urošević, D.; Pérez-Brito, D.; García-González, C.G. Variable neighbourhood search for bandwidth reduction. *Eur. J. Oper. Res.* **2010**, *200*, 14–27.
46. Pei, J.; Mladenović, N.; Urošević, D.; Brimberg, J.; Liu, X. Solving the traveling repairman problem with profits: A novel variable neighborhood search approach. *Inf. Sci.* **2020**, *507*, 108–123.
47. Xiao, Y.; Zhao, Q.; Kaku, I.; Mladenović, N. Variable neighbourhood simulated annealing algorithm for capacitated vehicle routing problems. *Eng. Optim.* **2014**, *46*, 562–579.
48. Salehipour, A.; Sepehri, M.M. Exact and heuristic solutions to minimize total waiting time in the blood products distribution problem. *Adv. Oper. Res.* **2012**, *2012*, 393890.
49. Zhou, Y.; Xu, Y.; Xie, K.; Li, J. Joint optimization of storage allocation and picking efficiency for fresh products using a particle swarm-guided hybrid genetic algorithm. *Mathematics* **2025**, *13*, 3428.
50. Drezner, Z.; Brimberg, J.; Mladenović, N.; Salhi, S. New heuristic algorithms for solving the planar p-median problem. *Comput. Oper. Res.* **2015**, *62*, 296–304.
51. Irawan, C.A.; Salhi, S.; Chan, H.K. A continuous location and maintenance routing problem for offshore wind farms: Mathematical models and hybrid methods. *Comput. Oper. Res.* **2022**, *144*, 105825.
52. Kocatürk, F.; Tütüncü, G.Y.; Salhi, S. The multi-depot heterogeneous VRP with backhauls: Formulation and a hybrid VNS with GRAMPS meta-heuristic approach. *Ann. Oper. Res.* **2021**, *307*, 277–302.
53. Kong, M.; Liu, X.; Pei, J.; Pardalos, P.M.; Mladenović, N. Parallel-batching scheduling with nonlinear processing times on a single and unrelated parallel machines. *J. Glob. Optim.* **2020**, *78*, 693–715.

54. Amaldass, N.I.L.; Lucas, C.; Mladenović, N. A heuristic hybrid framework for vector job scheduling. *Yugosl. J. Oper. Res.* **2017**, *27*, 31–45.
55. Simeonova, L.; Wassan, N.; Salhi, S.; Nagy, G. The heterogeneous fleet vehicle routing problem with light loads and overtime: Formulation and population variable neighbourhood search with adaptive memory. *Expert Syst. Appl.* **2018**, *114*, 183–195.
56. Bouzid, M.C.; Haddadene, H.A.; Salhi, S. An integration of Lagrangian split and VNS: The case of the capacitated vehicle routing problem. *Comput. Oper. Res.* **2017**, *78*, 513–525.
57. Irawan, C.A.; Salhi, S.; Drezner, Z. Hybrid meta-heuristics with VNS and exact methods: Application to large unconditional and conditional vertex p-centre problems. *J. Heuristics* **2016**, *22*, 507–537.
58. Daza-Escorcia, J.M.; Álvarez-Martínez, D. A matheuristic approach based on variable neighborhood search for the static repositioning problem in station-based bike-sharing systems. *Mathematics* **2024**, *12*, 3573.
59. Lan, S.; Fan, W.; Yang, S.; Mladenović, N.; Pardalos, P.M. Solving a multiple-qualifications physician scheduling problem with multiple types of tasks by dynamic programming and variable neighborhood search. *J. Oper. Res. Soc.* **2022**, *73*, 2043–2058.
60. Palubeckis, G. A variable neighborhood search and simulated annealing hybrid for the profile minimization problem. *Comput. Oper. Res.* **2017**, *87*, 83–97.
61. Palubeckis, G.; Tomkevičius, A.; Ostreika, A. Hybridizing simulated annealing with variable neighborhood search for bipartite graph crossing minimization. *Appl. Math. Comput.* **2019**, *348*, 84–101.
62. Hansen, P.; Mladenović, N.; Moreno Pérez, J.A. Variable neighbourhood search: Methods and applications. *Ann. Oper. Res.* **2010**, *175*, 367–407.
63. Hansen, P.; Mladenović, N.; Todosijević, R.; Hanafi, S. Variable neighborhood search: Basics and variants. *EURO J. Comput. Optim.* **2017**, *5*, 423–454.
64. Kirkpatrick, S.; Gelatt, C.D.; Vecchi, M.P. Optimization by simulated annealing. *Science* **1983**, *220*, 671–680.
65. Černý, V. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *J. Optim. Theory Appl.* **1985**, *45*, 41–51.
66. Mladenović, N.; Hansen, P. Variable neighborhood search. *Comput. Oper. Res.* **1997**, *24*, 1097–1100.
67. Kusiak, A.; Wang, J. Concurrent engineering: Simplification of the design process. In Proceedings of the Fourth International IFIP TC5 Conference on Computer Applications in Production and Engineering–Integration Aspects (CAPE '91), Computer Applications in Production and Engineering, Bordeaux, France, 10–12 September 1991; Elsevier: New York, NY, USA, 1991; pp. 297–304.
68. Austin, S.A.; Baldwin, A.N.; Hassan, T.; Newton, A.J. Techniques for the management of information flow in building design. In *Information Technology in Civil and Structural Engineering Design*; Inverleith Spottiswoode: Edinburgh, UK, 1996; pp. 119–123.
69. Rutenbar, R.A. Simulated annealing algorithms: An overview. *IEEE Circuits Devices Mag.* **1989**, *5*, 19–26.
70. van Laarhoven, P.J.M. *Theoretical and Computational Aspects of Simulated Annealing*; Centrum voor Wiskunde en Informatica: Amsterdam, The Netherlands, 1988.
71. Wen, M.; Lin, J.; Qian, Y.; Huang, W. Scheduling interrelated activities in complex projects under high-order rework: A DSM-based approach. *Comput. Oper. Res.* **2021**, *130*, 105246.
72. Ebufegha, A.J.; Li, S. A hybrid algorithm for task sequencing problems with iteration in product development. *J. Oper. Res. Soc.* **2022**, *73*, 1549–1561.