mathematics

MDPI

Article

A Variable Neighborhood Search Approach for the Dynamic Single Row Facility Layout Problem

Gintaras Palubeckis *, Armantas Ostreika and Jūratė Platužienė

Faculty of Informatics, Kaunas University of Technology, Studentu 50-408, 51368 Kaunas, Lithuania; armantas.ostreika@ktu.lt (A.O.); jurate.platuziene@ktu.lt (J.P.)

* Correspondence: gintaras.palubeckis@ktu.lt or gintaras.palubeckis77@gmail.com

Abstract: The dynamic single row facility layout problem (DSRFLP) is defined as the problem of arranging facilities along a straight line during a multi-period planning horizon with the objective of minimizing the sum of the material handling and rearrangement costs. The material handling cost is the sum of the products of the flow costs and center-to-center distances between facilities. In this paper, we focus on metaheuristic algorithms for this problem. The main contributions of the paper are three-fold. First, a variable neighborhood search (VNS) algorithm for the DSRFLP is proposed. The main version of VNS uses an innovative strategy to start the search from a solution obtained by constructing an instance of the single row facility layout problem (SRFLP) from a given instance of the DSRFLP and applying a heuristic algorithm for the former problem. Second, a fast local search (LS) procedure is developed. The innovations of this procedure are two-fold: (i) the fast insertion and swap neighborhood exploration techniques are adapted for the case of the dynamic version of the SRFLP; and (ii) to reduce the computational time, the swap operation is restricted on pairs of facilities of equal lengths. Provided the number of planning periods is a constant, the neighborhood exploration procedures for n facilities have only $O(n^2)$ time complexity. The superiority of these procedures over traditional LS techniques is also shown by performing numerical tests. Third, computational experiments on DSRFLP instances with up to 200 facilities and three or five planning periods are carried out to validate the effectiveness of the VNS approach. The proposed VNS heuristic is compared with the simulated annealing (SA) method which is the state of the art algorithm for the DSRFLP. Experiments show that VNS outperforms SA by a significant margin.

Keywords: combinatorial optimization; facility layout; metaheuristics; variable neighborhood search; local search

MSC: 90B80; 90C27; 90C59; 68R05

Citation: Palubeckis, G.; Ostreika, A.; Platužienė, J. A Variable Neighborhood Search Approach for the Dynamic Single Row Facility Layout Problem. *Mathematics* **2022**, *10*, 2174. https://doi.org/10.3390/math10132174

Academic Editors: Adrian Deaconu, Petru Adrian Cotfas and Daniel Tudor Cotfas

Received: 9 May 2022
Accepted: 20 June 2022
Published: 22 June 2022

1. Introduction

The facility layout problems are concerned with finding an efficient arrangement of physical facilities (e.g., machines or manufacturing cells) on a planar site. An important member of this class of problems is the single row facility layout problem (SRFLP). It asks to arrange the facilities along a straight line so as to minimize the sum of the products of the flow costs and center-to-center distances between facilities [1]. The objective function of the SRFLP represents the material handling cost that is a good measure to evaluate the efficiency of a layout. A feasible solution is a permutation of facilities. It can be noted that the SRFLP is a static problem, because the material flows between facilities are assumed to be constant. This assumption, however, may not be always valid in practice. In a dynamic layout, the flows of material between facilities can change during a planning horizon. There are many factors that play a role in this, such as the change in the design of existing products, removing an existing product or adding a new product to the product line, varying product demand, shortening life cycle of products, the change in the sequence

of operations, and the introduction of new safety standards. Because of changes in the layout environment, a multi-period planning horizon is considered. The material flows between facilities can change from one planning period to the next. When minimizing the material handling cost, it may happen that the permutation of facilities in period t, $t > 1$, is different from the permutation of facilities in period $t-1$. In such a case, one or more facilities are moved (shifted) to new locations at the beginning of period t. The objective function of the dynamic facility layout problem consists of two parts: the material handling cost, and the rearrangement costs of facilities.

A dynamic version of the SRFLP, called the *dynamic single-row facility layout problem* (DSRFLP) was introduced by Şahin et al. [2]. In their problem formulation, the rearrangement cost for a facility occurs when the center coordinate of this facility in one planning period is different from that in the preceding or succeeding periods. Figure 1 shows a layout plan for a DSRFLP instance with seven facilities and three planning periods. Costs are incurred for the relocation of facilities 3, 4, and 5 at the beginning of period 2 and for the relocation of facilities 2 and 4 at the beginning of period 3. Notice that facilities 2 and 4 are of equal size and therefore, facility 5 is not moved in period 3.

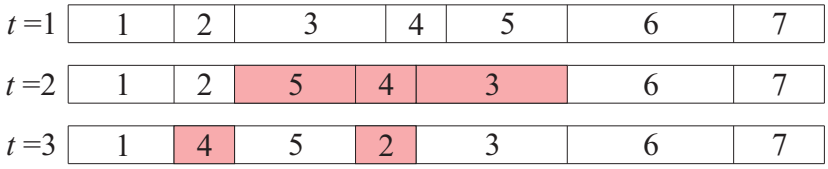

Figure 1. Example layout with three planning periods.

Let us denote the set of facilities by H, their number by n, and the number of planning periods by m. A feasible solution of the DSRFLP is an ordered collection $p = \{p_1, \ldots, p_m\}$ of m $n-$element permutations $p_t = (p_t(1), \ldots, p_t(n))$, $t = 1, \ldots, m$, where $p_t(i) \in H$, $i \in \{1, \ldots, n\}$, is the facility in the ith position during period t. We denote the set of all feasible solutions of the DSRFLP by Π^m. Let L_s, $s \in \{1, \ldots, n\}$, stand for the length of facility s. For convenience, the main notations used in this paper are presented in Table 1. Mathematically, the DSRFLP can be expressed as follows:

$$\min_{p \in \Pi^m} F(p) = \sum_{t=1}^{m} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} w_{tp_t(i)p_t(j)} d_t(p_t(i), p_t(j)) + \sum_{t=2}^{m} \sum_{s \in I(p_{t-1}, p_t)} r_{ts}, \tag{1}$$

where

$$d_t(p_t(i), p_t(j)) = L_{p_t(i)}/2 + \sum_{i<k<j} L_{p_t(k)} + L_{p_t(j)}/2, \tag{2}$$

$$w_{tp_t(i)p_t(j)} = \phi_{tp_t(i)p_t(j)} \psi_{p_t(i)p_t(j)}, \tag{3}$$

$\phi_{tp_t(i)p_t(j)}$ represents the material flow between facilities $p_t(i)$ and $p_t(j)$ during period t, $\psi_{p_t(i)p_t(j)}$ is the cost of transferring a unit of material per distance unit between facilities $p_t(i)$ and $p_t(j)$, r_{ts} is the rearrangement cost of facility s at the beginning of period t, and $I(p_{t-1}, p_t)$ is the set of facilities whose location during period t differs from that during period $t-1$ (in Figure 1, $I(p_1, p_2) = \{3, 4, 5\}$ and $I(p_2, p_3) = \{2, 4\}$). Equation (2) determines the distance between facilities, and Equation (3) gives the cost of material flow per distance unit between facilities.

Table 1. Main notations used in this paper.

Notation	Description
H	Set of facilities
n	Number of facilities ($n = \|H\|$)
m	Number of planning periods
s, u, v	Indices used for facilities ($s, u, v \in H$)
t	Index of planning periods
L_s	Length of facility s
ϕ_{tsu}	Material flow between facilities s and u in period t
ψ_{su}	Cost of transferring a unit of material per distance unit between facilities s and u
w_{tsu}	Material flow cost between facilities s and u in period t
x_{ts}	Center coordinate of facility s in period t
$d_t(s,u)$	Distance between centers of facilities s and u in period t ($d_t(s,u) = \|x_{ts} - x_{tu}\|$)
r_{ts}	Rearrangement cost of facility s at the beginning of period t
$I(p_{t-1}, p_t)$	Set of facilities whose location during period t differs from that during period $t-1$
w'_{su}	Material flow cost between facilities s and u in the SRFLP instance
λ^+_{uv}	Half-sum of lengths of facilities u and v
λ^-_{uv}	Half-difference between lengths of facilities u and v
p_t	Layout (permutation of facilities) during period t
$p = \{p_1, \ldots, p_m\}$	Solution (layout plan) of the DSRFLP
Π^m	Set of all feasible solutions
$F(p)$	Objective function of the DSRFLP
F^*	Objective function value of the best solution
p^*	Best solution
$\bar{F}$	Average objective function value
F_{avdev}	Average deviation of the objective function from the reference value
c_{tq}	Sum of flow costs between the first q facilities and the remaining $n-q$ facilities during period t
$G(u,t,x)$	Change in rearrangement cost incurred by placing facility u at location x during period t
$x_{ts}(l)$	Center coordinate of facility s when it is inserted at position l in permutation p_t
$\tilde{N}_z(p)$	Interchange neighborhood of depth z of solution p
$N(p)$	Insertion neighborhood of solution p
T_{lim}	Maximum time limit for algorithm execution

1.1. Related Work

If $t = 1$, then the second term in (1) vanishes, and the problems (1)–(3) becomes the SRFLP. Many algorithms for solving the SRFLP have been proposed. Several exact methods have been developed in recent years. They include mixed-integer linear programming [3], cutting plane algorithm [4], branch-and-cut [5], and semidefinite programming approaches [6,7]. The largest SRFLP instance solved to optimality is of size 42 [7]. To deal with larger sized SRFLP instances, one common option is to use metaheuristic-based algorithms. The more recent metaheuristic approaches available in the literature are tabu search [8,9], genetic algorithms [10–12], a Lin–Kernighan heuristic [13], hybrid estimation of distribution algorithm [14], scatter search [15], variable neighborhood search (VNS) [16], greedy randomized adaptive search procedure (GRASP) with path relinking [17], a hybrid algorithm of VNS and ant colony optimization [18], simulated annealing [19], a cross-entropy approach [20], a population-based improvement heuristic with local search [21], GRASP [22], a simplified swarm optimization algorithm [23], differential evolution [24], and algebraic differential evolution for permutations [25]. Sun et al. [26] used graphics processing units to solve the SRFLP with the two-opt-based simulated annealing algorithm. A review of the mathematical models and solution techniques of the SRFLP can be found in [1,27].

In the literature, there are several other facility layout problems that are akin to the SRFLP. One of them is the constrained SRFLP in which some facilities need to be placed

in certain locations or in specified orders. A permutation-based genetic algorithm [28] and a fireworks algorithm [29] were proposed for solving this problem. Keller [30] developed three construction heuristics for the SRFLP with machine-spanning clearances. Amaral [31] and Yang et al. [32] proposed mixed-integer programming models for the parallel row ordering problem. Several algorithms were presented to solve the corridor allocation problem, such as simulated annealing and tabu search [33], a permutation-based genetic algorithm hybridized with a local search technique [34], and a scatter search algorithm [35]. Recently, attention was attracted to the space-free multi-row facility layout problem. An exact method for this problem was presented in [36]. An efficient VNS algorithm for the same problem was developed in [37].

A mixed-integer linear programming model and solution approaches for the DSRFLP were proposed by Şahin et al. [2]. They used CPLEX to solve the linear model. However, the solver failed to produce a provably optimal solution for instances of size greater than 10. To find high quality solutions for larger DSRFLP instances, Şahin et al. [2] proposed two metaheuristic-based approaches: a genetic algorithm (GA) and a simulated annealing (SA) technique. Their GA is strengthened by integrating a restart strategy and applying the concept of acceptance probability. The proposed SA algorithm is also enhanced with a restart strategy. The authors reported computational results on 20 problem instances with up to 100 facilities and 3 or 5 planning periods. The empirical results demonstrated the superiority of the SA algorithm over the GA implementation.

There is a vast amount of literature related to the dynamic version of the facility layout problems. Gong et al. [38] proposed a hybrid algorithm of harmony search for the dynamic parallel row ordering problem. Their algorithm combines a harmony search technique with a tabu search heuristic. The authors presented the results of computational experiments on problem instances with up to 70 facilities. Guan et al. [39] proposed a hybrid evolution algorithm for solving a dynamic extended row facility layout problem. Both combinatorial and continuous aspects of the problem were taken into account. However, historically, most algorithms in the field were developed for the dynamic facility layout problem (DFLP) in which the layout of each timeframe is modeled as a quadratic assignment problem. The first such algorithms (dynamic programming-based optimal and heuristic procedures) were proposed by Rosenblatt [40]. Later, many metaheuristic-based algorithms for the DFLP were reported. Balakrishnan and Cheng [41] developed a genetic algorithm for solving this problem. A hybrid GA for the DFLP was proposed in [42]. A simulated annealing heuristic for this problem was presented in [43]. Ant colony optimization algorithms for the DFLP were developed in [44,45]. Hybrid ant system heuristics were proposed in [46]. Şahin and Türkbey [47] presented an algorithm-hybridizing SA with tabu search. Three new tabu search heuristics for the DFLP were developed by McKendall and Liu [48]. Hosseini-Nasab and Emami [49] designed a hybrid particle swarm optimization algorithm to solve the DFLP. Turanoğlu and Akkaya [50] proposed a hybrid algorithm combining SA and a bacterial foraging optimization technique. The results of the experimental comparison of a number of different algorithms from the literature were provided in a recent study on the DFLP by Zouein and Kattan [45]. A review of the recent advances on the DFLP can be found in [51]. The extensive literature on both static and dynamic versions of the facility layout problems was reviewed in [52].

1.2. Our Contribution

The analysis of literature shows that there has been considerable interest in developing algorithms for the dynamic facility layout problems. One of such problems is the DSRFLP. However, research on the DSRFLP is in its early stages. Considering this observation, our motivation is to develop a reasonably fast heuristic algorithm that could perform well on large DSRFLP instances. Our algorithm is based on the VNS metaheuristic. One purpose of this paper is to design a fast local search (LS) procedure. It takes only $O(1)$ time to compute the gain of a swap or insertion operation. The procedure plays a central role within the

VNS framework. Our specific goal is to compare the performance of the VNS technique with that of the SA algorithm proposed in [2].

The main contributions of this paper are summarized as follows:

- A variable neighborhood search algorithm to solve the DSRFLP. It is one of the first heuristic approaches proposed to deal with this new problem. The approach uses an innovative strategy to start the search from a solution obtained by constructing an instance of the SRFLP from a given instance of the DSRFLP and applying a heuristic algorithm for the former problem. A simpler version of VNS uses a random permutation of facilities as a starting solution.
- A fast LS procedure. The innovations of the proposed procedure are two-fold: (a) the fast insertion and swap neighborhood exploration techniques are adapted for the case of the dynamic version of the SRFLP; and (b) to reduce the computational time, the swap operation is restricted on pairs of facilities of equal lengths. The superiority of the fast neighborhood exploration procedures over traditional LS techniques is shown by performing numerical tests. The importance of the proposed LS method goes beyond VNS: it may be considered a useful ingredient for designing other metaheuristic algorithms for the DSRFLP.
- Numerical experimentation on DSRFLP instances of size up to 200 to validate the effectiveness of the VNS approach. The two versions of VNS are compared with the SA algorithm of Şahin et al. [2]. Experiments show the excellent performance of the VNS version that starts with a heuristically constructed initial solution. This VNS implementation outperforms SA by a significant margin.
- Preparation of a set of publicly available DSRFLP instances. Experiments show that larger instances in this set are very difficult for both VNS and SA algorithms. To find improved solutions, a large amount of CPU time may be required. These instances may be used for evaluating future approaches to dynamic single row facility layout.

The rest of the paper is organized as follows. In the next section, we present our VNS approach. In Section 3, we give a detailed description of our LS procedure. Section 4 is devoted to the experimental analysis and comparisons of algorithms. Section 5 provides an empirical analysis of LS variants. Concluding remarks are given in Section 6.

2. Variable Neighborhood Search

The variable neighborhood search method is a metaheuristic that has been shown to be very successful in solving many combinatorial optimization problems. The basic idea of VNS is the systematic change of a neighborhood combined with solution perturbation and local search procedures. During algorithm execution, the neighborhood of a solution is explored using a set of predefined neighborhood structures. Since its introduction in 1997 [53], VNS has undergone various modifications and enhancements. A discussion of the basic concepts and successful applications of VNS can be found in survey papers [54].

Before presenting our algorithm, we need to define neighborhood structures used in the search process. Let $p = (p_1, \ldots, p_m) \in \Pi^m$ represent a solution of the DSRFLP. For $z \in \{1, \ldots, z_{\max}\}$, we denote by $\tilde{N}_z(p)$ the set of all solutions that can be obtained from p by performing z pairwise interchanges of facilities subject to the condition that each facility changes its position in each permutation p_t $t \in \{1, \ldots, m\}$ at most once. The neighborhood structures $\{\tilde{N}_z(p) \mid p \in \Pi^m\}, z = 1, \ldots, z_{\max}$, are appropriate in cases where a permutation-based combinatorial optimization problem is solved [55–57]. We use these structures in the shaking (solution perturbation) step of the algorithm.

The steps of our implementation of the VNS method are given in Algorithm 1. The algorithm starts by generating an initial solution, either randomly or by a heuristic procedure. This step will be discussed later in this section. The initial solution is improved by applying a local search procedure LS (Line 2). We defer the description of LS to the next section. The main part of VNS is the "while" loop (Lines 4–14) which executes until the time condition is satisfied. The search is terminated when the maximum time limit, T_{lim}, is reached. The algorithm has three parameters that control the search process. The

parameter $z_{\min}$ determines the size of the neighborhood the search begins from. The largest possible size of the neighborhood is defined by $z_{\max} = \rho n$, where ρ is another parameter of the algorithm. The variable z_{step} is used to move from the current neighborhood to the next one. We set $z_{\text{step}} = \max(\lfloor z_{\max}/\theta \rfloor, 1)$, where the scaling factor $\theta > 0$ is the third parameter of VNS. The best solution to the algorithm is denoted as p^* and its value is f^*. The inner "while" loop of the algorithm iterates over the following three steps: the perturbation of the best solution p^* (procedure `shake` in Line 7), local search (Line 8), and the selection of the size of the next neighborhood (procedure `neighborhood_change` in Line 9). These steps are executed until z becomes greater than $z_{\max}$. At the end of each iteration, the termination condition of VNS is checked (Line 10).

Algorithm 1 Variable neighborhood search

VNS($z_{\min}, z_{\max}, z_{\text{step}}, T_{\text{lim}}$)
Input: Instance of the DSRFLP, parameters $z_{\min}, z_{\max}, z_{\text{step}}$, and T_{lim}.
Output: Best found solution p^* and its value f^*.
// $z_{\min}$, $z_{\max}$, and z_{step} control the size of the neighborhood
// T_{lim} is the time limit for VNS
1: Generate an initial solution $p \in \Pi^m$
2: $f :=$ LS(p)
3: Assign p to p^* and f to f^*
4: **while** time limit T_{lim} not reached **do**
5: $z := z_{\min}$
6: **while** $z \leqslant z_{\max}$ **do**
7: $p :=$ `shake`(p^*, z)
8: $f :=$ LS(p)
9: $z :=$ `neighborhood_change`($p, p^*, f, f^*, z, z_{\min}, z_{\text{step}}$)
10: **if** elapsed time is more than T_{lim} **then**
11: Break from the loop
12: **end if**
13: **end while**
14: **end while**
15: Stop with the solution p^* of value f^*

The pseudo-code of the shaking procedure and the neighborhood change function is given in Algorithms 2 and 3, respectively. The aim of `shake` is to perturb the best solution p^* (or, more precisely, its copy p). The parameter z is the total number of pairwise interchange moves that are executed by the procedure. The number of moves performed on the permutation p_t, $t \in \{1, \ldots, m\}$, is denoted by q_t. The procedure uniformly and randomly chooses an integer in the interval $[1, m]$ z times. The value of q_t is equal to the number of times that the integer t is selected. Further steps of the procedure (Lines 3–10) are executed for each planning period t with $q_t > 0$. The inner loop (Lines 5–9) starts by randomly selecting two facilities from the set $\{p_t(i) \mid i \in I\}$. They are denoted as $p_t(k)$ and $p_t(l)$. Then, the permutation p_t is updated by swapping the positions of the selected facilities. The role of the set I in the algorithm is to guarantee that each facility is selected at most once. The solution returned by `shake` belongs to the neighborhood $\tilde{N}_z(p^*)$. The procedure `neighborhood_change` either increases the shaking parameter z by z_{step} or sets it to the minimum value $z_{\min}$ if an improved solution has been found. The procedure is responsible for memorizing this solution (Line 2).

Algorithm 2 Shake function

`shake`(p^*, z)
Input: Best-so-far solution p^*, parameter z.
Output: Solution p in the neighborhood of p^*.
1: Assign p^* to p
2: Randomly split z into m nonnegative numbers q_t, $t = 1, \ldots, m$
3: **for** $t = 1, \ldots, m$ such that $q_t > 0$ **do**
4: $I := H$
5: **for** q_t times **do**
6: Randomly select k and $l \neq k$ from I
7: Swap positions of facilities $p_t(k)$ and $p_t(l)$ in p_t
8: $I := I \setminus \{k, l\}$
9: **end for**
10: **end for**
11: **return** $p = (p_1, \ldots, p_m)$

Algorithm 3 Neighborhood change function

`neighborhood_change`$(p, p^*, f, f^*, z, z_{\min}, z_{\text{step}})$
Input: Current solution p of value f, best-so-far solution p^* of value f^*, parameters $z_{\min}$ and z_{step}.
Output: Possibly updated p^* and f^*, parameter z.
1: **if** $f < f^*$ **then**
2: Assign p to p^* and f to f^*
3: $z := z_{\min}$
4: **else**
5: $z := z + z_{\text{step}}$
6: **end if**
7: **return** z

Now, we return to the question of generating an initial solution to the DSRFLP. A simple way is to randomly generate a permutation of n facilities and assign this permutation to p_t for each $t \in \{1, \ldots, m\}$. We call this configuration of our algorithm VNS1. An alternative strategy is to apply a heuristic technique. Our choice in this work is to use a VNS algorithm proposed in [16] for solving the SRFLP. We obtain an instance of the SRFLP with the flow matrix $W' = (w'_{su})$ from an instance of the DSRFLP with a set of flow matrices $W_t = (w_{tsu})$, $t = 1, \ldots, m$, straightforwardly by summing the matrices W_t. Formally, $w'_{su} = \sum_{t=1}^{m} w_{tsu}$, $s, u = 1, \ldots, n$. Like VNS presented in this paper, the algorithm in [16] is equipped with a CPU time-based stopping criterion. Therefore, we have to share the time resources between the generation of an initial solution (Line 1 of Algorithm 1) and the remaining steps of the VNS. We set the cutoff time for the first step (Line 1) to βT_{lim}. A suitable value of the parameter β should be chosen experimentally. Intuitively, β is expected to be less than 0.1. An experiment for selecting β is described in Section 4. Assume that $\tilde{p}$ is the solution produced by the algorithm in [16] for the SRFLP instance with the flow matrix W'. We constructed an initial solution for the DSRFLP by setting $p_t = \tilde{p}$ for each $t = 1, \ldots, m$. We refer to this configuration of VNS as VNS2. It can be noticed that VNS1 is obtained from VNS2 by taking $\beta = 0$.

3. Local Search

An important issue in the design of local search algorithms is the choice of a neighborhood structure or structures. Our first priority is to use the insertion neighborhood. Let us denote this neighborhood for $p = (p_1, \ldots, p_m) \in \Pi^m$ by $N(p)$. It consists of all solutions that can be obtained from p by removing a facility from its current position in a permutation p_t, $t \in \{1, \ldots, m\}$, and inserting it at a different position in the same

permutation. Given $p \in \Pi^m$, the construction of a solution in the neighborhood $N(p)$ is called an insertion move. As noted by Schiavinotto and Stützle [55], insertion move is one of the basic operators in permutation-based optimization problems. Assume that $p' = (p'_1, \ldots, p'_m) \in N(p)$ is obtained from p by removing facility $s = p_t(k)$ from position k and inserting it at position l. The change in cost between p and p' is called the move gain. We denote it by $\delta(p, k, l) = F(p') - F(p)$. The insertion operation is illustrated in Figure 2 for $n = 6$ and $m = 3$. Facility 1 is moved at the beginning of planning period 2 from its current position 3 to position 5. As a result, facilities 2 and 6 are also relocated.

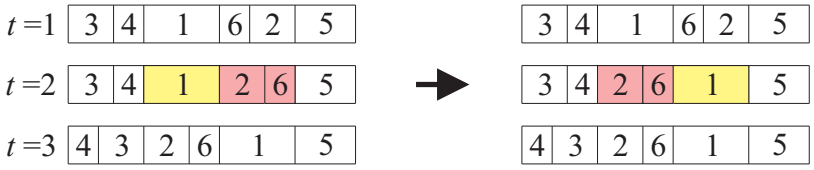

Figure 2. Relocating facility 1 from position 3 to position 5 during planning period 2.

To describe methods for computing $\delta(p, k, l)$, we need some notations. We denote by $X = (x_{tu})$ an $m \times n$ matrix whose entry x_{tu} is the center coordinate of facility u during planning period t. We also define the following two functions:

$$g_1(u,t,x) = \begin{cases} r_{tu} & \text{if } x_{tu} = x_{t-1,u} \neq x \\ -r_{tu} & \text{if } x_{tu} \neq x_{t-1,u} = x \\ 0 & \text{otherwise,} \end{cases} \tag{4}$$

$$g_2(u,t,x) = \begin{cases} r_{t+1,u} & \text{if } x_{tu} = x_{t+1,u} \neq x \\ -r_{t+1,u} & \text{if } x_{tu} \neq x_{t+1,u} = x \\ 0 & \text{otherwise.} \end{cases} \tag{5}$$

We note that x_{tu}, $x_{t-1,u}$, and $x_{t+1,u}$ are original positions of facility u in periods t, $t-1$, and $t+1$, respectively, and x is the position of facility u after the movement during period t. Clearly, (4) is defined for $t \in \{2, \ldots, m\}$ and (5) is defined for $t \in \{1, \ldots, m-1\}$. Let us consider the general case of $t \in \{2, \ldots, m-1\}$. Assume that the center coordinate of facility u changes from x_{tu} to x when u is moved to a new location at the beginning of period t. Then, the sum $G(u,t,x) = g_1(u,t,x) + g_2(u,t,x)$ expresses the change in rearrangement cost of p' incurred by this move operation. In Figure 2, this sum is $r_{2,1} - r_{3,1}$ for facility 1, $-r_{3,2}$ for facility 2, and $-r_{3,6}$ for facility 6. The above-defined sum reduces to a single term $G(u,t,x) = g_2(u,t,x)$ for $t = 1$ and $G(u,t,x) = g_1(u,t,x)$ for $t = m$.

Now we return to the computation of $\delta(p, k, l)$. Let $\delta'(p, k, l)$ denote the change in material flow cost incurred by the insertion move producing solution p' from the solution p. For our purposes, it is convenient to write the move gain as

$$\delta(p,k,l) = \delta'(p,k,l) + G(s,t,x_{ts}(l)), \tag{6}$$

where $s = p_t(k)$ as before and $x_{ts}(l)$ stands for the center coordinate of facility s in the layout during period t, which is obtained by inserting s at position l in the permutation p_t. The reason for using (6) is to compute both terms at the right-hand side of (6) (more precisely, $\delta'(p, k, l)$ and $x_{ts}(l)$) recursively.

To proceed, suppose that $l < k$. We note that the insertion move can be reduced to a sequence of elementary moves in which facility s is interchanged with its left neighbor. The insertion process starts by interchanging s with facility $p_t(k-1)$. Next s is interchanged with $p_t(k-2)$, then with $p_t(k-3)$, etc. Eventually, this process is going to end when facility s reaches position l in the permutation p_t. Let us focus on the last step in the described process: facility s is interchanged with facility $v = p_t(l)$. At this point, $\delta'(p, k, l+1)$ and $x_{ts}(l+1)$ are assumed to be known. The last step of the insertion process is illustrated in Figure 3. We see that after swapping the positions of facilities s and v, the distance between facility $p_t(i) = u$, $i < l$, and s decreases by L_v and that between u and v increases by L_s.

Similarly, the distance between facility $p_t(i)$, $i > l$, $i \neq k$, and s increases by L_v and that between $p_t(i)$ and v decreases by L_s. Based on these observations, we can write

$$\delta'(p,k,l) = \delta'(p,k,l+1) + \sum_{i=1}^{l-1}(w_{tp_t(i)v}L_s - w_{tp_t(i)s}L_v) + \sum_{i=l+1,i\neq k}^{n}(w_{tp_t(i)s}L_v - w_{tp_t(i)v}L_s) + G(v,t,x_{tv}+L_s). \quad (7)$$

The last term in (7) represents the change in the cost resulting from the rearrangement of facility v. The new center coordinate of facility s is computed as follows:

$$x_{ts}(l) = x_{ts}(l+1) - L_v. \quad (8)$$

The initial conditions of the recurrence relations (7) and (8) are $\delta'(p,k,k) = 0$ and $x_{ts}(k) = x_{ts}$, respectively.

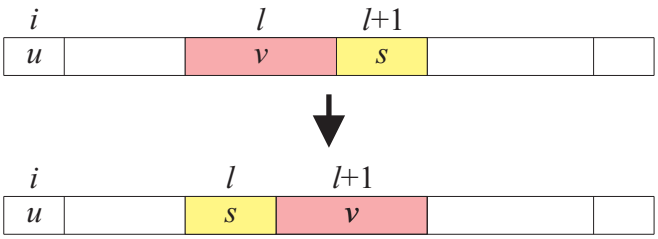

Figure 3. Interchanging facility s with its left neighbor v.

If $k < l$, then (7) and (8) are replaced by the following equations:

$$\delta'(p,k,l) = \delta'(p,k,l-1) + \sum_{i=1,i\neq k}^{l-1}(w_{tp_t(i)s}L_v - w_{tp_t(i)v}L_s) + \sum_{i=l+1}^{n}(w_{tp_t(i)v}L_s - w_{tp_t(i)s}L_v) + G(v,t,x_{tv}-L_s), \quad (9)$$

$$x_{ts}(l) = x_{ts}(l-1) + L_v. \quad (10)$$

The initial conditions of (9) and (10) are the same as in the case of $l < k$.

The approach we have just described, however, is not very efficient. It takes $O(n)$ time to compute $\delta'(p,k,l)$ using (7) or (9). We adopt a method that was proposed in [16] for solving the SRFLP. We use an $m \times (n-1)$ matrix $C = (c_{tq})$ with entries $c_{tq} = \sum_{i=1}^{q}\sum_{j=q+1}^{n} w_{tp_t(i)p_t(j)}$, $t = 1,\ldots,m$, $q = 1,\ldots,n-1$. Its entry c_{tq} represents the sum of flow costs between the first q facilities and the remaining $n-q$ facilities during period t. To compute C, we use two other matrices $E^1 = (e^1_{tu})$ and $E^2 = (e^2_{tu})$ of size $m \times n$, in which $e^1_{tu} = \sum_{i=1}^{q-1} w_{tp_t(i)u}$, $e^2_{tu} = \sum_{i=q+1}^{n} w_{tp_t(i)u}$, and where it is assumed that $u = p_t(q)$. The rows of the matrices E^1 and E^2 are indexed by periods and the columns by facilities. It is not difficult to see that

$$c_{tq} = c_{t,q-1} + e^2_{tp_t(q)} - e^1_{tp_t(q)}, t = 1,\ldots,m, q = 1,\ldots,n-1, \quad (11)$$

where by convention $c_{t0} = 0$, $t = 1,\ldots,m$. From C, we can build the matrix $C^- = (c^-_{tq})$ with entries $c^-_{tq} = c_{tq} - c_{t,q-1}$ and the matrix $C^+ = (c^+_{tq})$ with entries $c^+_{tq} = c_{tq} + c_{t,q-1}$, where $t = 1,\ldots,m$ and $q = 1,\ldots,n$. In these definitions, it is assumed that $c_{tn} = 0$, $t = 1,\ldots,m$. An efficient way to compute $\delta'(p,k,l)$ is provided by the following formulas.

Proposition 1. *Let $p \in \Pi^m$, $t \in \{1,\ldots,m\}$, $k \in \{1,\ldots,n\}$, and $s = p_t(k)$. Then, for $l = k-1, k-2, \ldots, 1$,*

$$\delta'(p,k,l) = \delta'(p,k,l+1) + L_s(w_{tsv} - c^-_{tl}) + L_v(\alpha_l + \alpha_{l+1} + c^-_{tk}) + G(v,t,x_{tv}+L_s), \quad (12)$$

where $v = p_t(l)$, $\alpha_l = \alpha_{l+1} + w_{tsv}$, *and initially* $\delta'(p,k,k) = 0$ *and* $\alpha_k = 0$.

For $l = k+1, k+2, \ldots, n$,

$$\delta'(p,k,l) = \delta'(p,k,l-1) + L_s(w_{tsv} + c^-_{tl}) + L_v(\alpha_{l-1} + \alpha_l - c^-_{tk}) + G(v,t,x_{tv} - L_s), \quad (13)$$

where $v = p_t(l)$, $\alpha_l = \alpha_{l-1} + w_{tsv}$, *and the initial conditions are the same as in (12).*

Proof. Consider the case of $l < k$. Suppose that the facility s is interchanged with its current left neighbor $v = p_t(l)$. Let the resulting change in material handling cost be denoted by $\delta'_1(p,k,l)$ and the cost of rearranging facility v be denoted by $\delta'_2(p,k,l)$. Thus, $\delta'(p,k,l) = \delta'(p,k,l+1) + \delta'_1(p,k,l) + \delta'_2(p,k,l)$. Based on Proposition 6 in [16], it can be concluded that $\delta'_1(p,k,l)$ is equal to $L_s(w_{tsv} - c^-_{tl}) + L_v(\alpha_l + \alpha_{l+1} + c^-_{tk})$. Moreover, from the definition of the functions g_1, g_2, and G it is obvious that $\delta'_2(p,k,l) = G(v,t,x_{tv} + L_s)$. Taken together, these two observations prove (12). The same line of reasoning applies to Equation (13). □

When facility s is relocated from position k to position l in the permutation p_t, the row of the matrix C^- corresponding to period t needs to be updated. Suppose first that $l < k$. Then, for each facility $v = p_t(q)$, $q \in \{l, \ldots, k-1\}$, the material flow cost w_{tsv} is added to e^1_{tv} and e^2_{ts} and subtracted from e^2_{tv} and e^1_{ts}. If $l > k$, then, for each facility $v = p_t(q)$, $q \in \{k+1, \ldots, l\}$, w_{tsv} is added to e^2_{tv} and e^1_{ts} and subtracted from e^1_{tv} and e^2_{ts}. This step of the algorithm also includes updating the tth row of the matrix X and the permutation p_t. Having updated E^1 and E^2, the new entries in the tth row of C are computed using (11). This is only performed for $q = \min(k,l), \ldots, \max(k,l) - 1$. Finally, the tth row of the matrix C^- is updated according to the definition of C^-.

To make the search more productive, our LS algorithm also applies the swap operator. It consists of swapping the positions of two facilities. The obtained solution belongs to the neighborhood $\tilde{N}_1$ defined in Section 2. To lessen the computational burden, the algorithm employs a reduced swap neighborhood $\tilde{N}'_1$. For $p \in \Pi^m$, a solution $p' \in \tilde{N}_1(p)$ belongs to $\tilde{N}'_1(p)$ if and only if it is obtained by interchanging in p_t, $t \in \{1, \ldots, m\}$, two facilities of equal length. Let these facilities be denoted by $s = p_t(k)$ and $u = p_t(l)$. Assume w.l.o.g. that $k < l$. Since $L_s = L_u$, the center coordinates of facilities $p_t(i)$, $i = k+1, \ldots, l-1$, in p' are the same as in p. This fact significantly reduces the complexity of computing the difference between $F(p')$ and $F(p)$. We call this difference the gain of the swap move, and denote it by $\Delta(p,k,l) = F(p') - F(p)$. One possible method to compute $\Delta(p,k,l)$ uses a distance matrix. Let us denote this $n \times n$ matrix by $D_t = (d_t(p_t(i), p_t(j)))$ where $d_t(p_t(i), p_t(j))$ is the distance between the centers of facilities $p_t(i)$ and $p_t(j)$ during period t as defined by Equation (2). Assume that the positions of facilities s and u with $L_s = L_u$ are swapped in the permutation p_t. This operation changes the material flow cost between each facility $v \in H \setminus \{s,u\}$ and facilities s and u. This change is equal to $w_{tvs}d_t(v,u) - w_{tvs}d_t(v,s) + w_{tvu}d_t(v,s) - w_{tvu}d_t(v,u) = (w_{tvs} - w_{tvu})(d_t(v,u) - d_t(v,s))$. Therefore, we can write

$$\Delta(p,k,l) = \sum_{v=1, v\neq s, v\neq u}^{n} (w_{tvs} - w_{tvu})(d_t(v,u) - d_t(v,s)) + G(s,t,x_{tu}) + G(u,t,x_{ts}). \quad (14)$$

The described method is simple but not very efficient. A good alternative is to adopt an approach proposed in [16]. To present an expression for $\Delta(p,k,l)$, we use the following matrices: $\lambda^+ = (\lambda^+_{uv} = (L_u + L_v)/2)$, $\lambda^- = (\lambda^-_{uv} = (L_u - L_v)/2)$, $u, v \in H$, $A_t = (a_{tvj})$, $v \in H$, $j = 1, \ldots, n$, $t \in \{1, \ldots, m\}$, with entries

$$a_{tvj} = \begin{cases} \sum_{i=j}^{q-1} w_{tvp_t(i)} & \text{if } j < q \\ \sum_{i=q+1}^{j} w_{tvp_t(i)} & \text{if } j > q \\ 0 & \text{if } j = q \end{cases}$$

for $v = p_t(q)$, and $B_t = (b_{tvj})$, $v \in H$, $j = 1, \ldots, n$, $t \in \{1, \ldots, m\}$, with entries

$$b_{tvj} = \begin{cases} a_{tvj}\lambda^+_{vp_t(j)} + \sum_{i=j+1}^{q-1} a_{tvi}\lambda^+_{p_t(i-1)p_t(i)} & \text{if } j < q \\ a_{tvj}\lambda^+_{vp_t(j)} + \sum_{i=q+1}^{j-1} a_{tvi}\lambda^+_{p_t(i)p_t(i+1)} & \text{if } j > q \\ 0 & \text{if } j = q \end{cases}$$

for $v = p_t(q)$. Matrices C^- and C^+ were already defined earlier in this section. With these matrices, we can provide a formula to calculate the gain $\Delta(p, k, l)$.

Proposition 2. *For $p \in \Pi^m$, $t \in \{1, \ldots, m\}$, $k \in \{1, \ldots, n-1\}$, $l \in \{k+1, \ldots, n\}$, $s = p_t(k)$, and $u = p_t(l)$,*

$$\Delta(p, k, l) = (c^-_{tl} - c^-_{tk} + 2w_{tsu})d_t(s, u) + 2(b_{ts,l-1} + b_{tu,k+1}) + \lambda^-_{su}(c^+_{tl} - c^+_{tk}) + G(s, t, x_{tu}) + G(u, t, x_{ts}). \quad (15)$$

Equation (15) follows from Proposition 2 in [16] and the definition of function G. If $l = k + 1$, then (15) reduces to the following equation

$$\Delta(p, k, l) = (c^-_{tl} - c^-_{tk} + 2w_{tsu})d_t(s, u) + \lambda^-_{su}(c_{tl} - c_{t,k-1}) + G(s, t, x_{tu}) + G(u, t, x_{ts}). \quad (16)$$

Before starting to use (15) and (16), a number of matrices, including B_t, $t = 1, \ldots, m$, need to be initialized. To build B_t, first the matrix A_t should be computed. Consider facility $v = p_t(q)$. By definition, $a_{tvq} = 0$. Other entries in the vth row of A_t can be iteratively obtained by setting

$$a_{tvj} = a_{tv,j-1} + w_{tvp_t(j)}, j > q, \quad (17)$$

$$a_{tvj} = a_{tv,j+1} + w_{tvp_t(j)}, j < q. \quad (18)$$

The corresponding row of the matrix B_t can be constructed by starting with $b_{tvq} = 0$ and applying the following equations:

$$b_{tvj} = b_{tv,j-1} + a_{tv,j-1}\lambda^-_{vp_t(j)} + a_{tvj}\lambda^+_{vp_t(j)}, j > q, \quad (19)$$

$$b_{tvj} = b_{tv,j+1} + a_{tv,j+1}\lambda^-_{vp_t(j)} + a_{tvj}\lambda^+_{vp_t(j)}, j < q. \quad (20)$$

The proof of (17)–(20) can be found in [16].

Suppose that facilities $s = p_t(k)$ and $u = p_t(l)$, $l > k$, are interchanged in the permutation p_t. Then, the matrices A_t and B_t need to be updated. This can be easily done using Equations (17)–(20). For simplicity reasons, we assume that p in these equations refers to the solution obtained after performing the swap move. Let $v = p_t(q)$. If $q < k$, then (17) is used for $j = k, \ldots, l-1$ and (19) for $j = k, \ldots, n$. If $q > l$, then (18) is used for $j = l, l-1, \ldots, k+1$ and (20) for $j = l, l-1, \ldots, 1$. If $k < q < l$, then A_t is updated by Equation (17) for $j = l, \ldots, n$ and by (18) for $j = k, k-1, \ldots, 1$. Similarly, B_t is updated by Equation (19) for $j = l, \ldots, n$ and by (20) for $j = k, k-1, \ldots, 1$. Finally, if $q = k$ (in this case, $v = u$) or $q = l$ (in this case, $v = s$), the vth row of A_t and B_t is created anew by first setting $a_{tvq} = b_{tvq} = 0$ and then applying Equations (17)–(20).

Our local search algorithm for the DSRFLP explores the reduced swap neighborhood $\tilde{N}'_1$ and the insertion neighborhood N repeatedly. This fact also implies the need to update the matrices A_t and B_t after performing an insertion move. Like in the case of pairwise interchange of facilities, the matrices are updated using Equations (17)–(20). Assume that facility s is moved from position k to position l in permutation p_t. Let $i = \min(k, l)$, $i' = \max(k, l)$, and consider facility $v = p_t(q)$ (here, p_t stands for the updated permutation). If $q < i$, then a_{tvj}, $j = i, \ldots, i'-1$, are calculated by Equation (17) and b_{tvj}, $j = i, \ldots, n$, are calculated by Equation (19). Other entries in the vth row of A_t and B_t remain unchanged. If $q > i'$, then (18) is used for $j = i', i'-1, \ldots, i+1$ and (20) for $j = i', i'-1, \ldots, 1$. If $q = l$ (in this case, $v = s = p_t(l)$), then first a_{tvl} as well as b_{tvl} are set to 0 and then Equations (17)–(20)

are applied. Consider now the remaining case where $i \leqslant q \leqslant i'$ and $q \neq l$. If $k < l$, then a_{tvj} is set to $a_{tvj} = a_{tv,j+1}$ and b_{tvj} is set to $b_{tvj} = b_{tv,j+1}$ for $j = k, \ldots, l-1$. If $k > l$, then a_{tvj} is set to $a_{tvj} = a_{tv,j-1}$ and b_{tvj} is set to $b_{tvj} = b_{tv,j-1}$ for $j = k, k-1, \ldots, l+1$. Moreover, a_{tvj} is calculated by (17) and b_{tvj} by (19) for $j = j', \ldots, n$, where $j' = l$ if $k < l$ and $j' = k+1$ if $k > l$. Furthermore, a_{tvj} is calculated by (18) and b_{tvj} by (20) for $j = j'', j''-1, \ldots, 1$, where $j'' = k-1$ if $k < l$ and $j'' = l$ if $k > l$. The step-by-step routines to update the matrices A_t and B_t can be found in [16].

Algorithm 4 gives the pseudocode of the LS component of the approach. The input to LS includes the solution p from which the search is started. This solution is possibly replaced by a better one during the search process. The returned solution p is locally optimal with respect to two neighborhoods, the reduced swap neighborhood $\tilde{N}_1'(p)$ and the insertion neighborhood $N(p)$. The variable f stores the objective function value of the solution p. The algorithm starts with the initialization of the matrices C, C^-, C^+, X, D_t, and B_t, $t = 1, \ldots, m$. Since B_t depends on the matrix A_t, the latter, for each $t \in \{1, \ldots, m\}$, is initialized as well. Then, the algorithm proceeds iteratively. At each iteration, it first repeatedly explores the reduced swap neighborhood $\tilde{N}_1'(p)$ of the current solution p until a locally optimal solution is reached (Lines 4–8). Then, the procedure `explore_insertion_neighborhood` is triggered. It either returns an improved solution (if $\Delta^* < 0$) or says that the solution p is locally optimal with respect to the neighborhood $N(p)$ (if $\Delta^* = 0$). In the latter case, the algorithm terminates. We apply the `explore_swap_neighborhood` procedure first and `explore_insertion_neighborhood` second because the number of possible swap moves is expected to be much less than the number of possible insertion moves. This is because the swap operation is restricted on pairs of facilities with equal lengths.

Algorithm 4 Local search

LS(p)
Input: Solution p.
Output: Possibly improved solution p and its value f.
1: Initialize data (matrices C, C^-, C^+, X, D_t, and B_t, $t = 1, \ldots, m$)
2: Compute $f = F(p)$ and set $\mu :=$ `true`
3: **while** $\mu =$ `true` **do**
4: $\Delta^* := -1$
5: **while** $\Delta^* < 0$ **do**
6: $\Delta^* :=$ `explore_swap_neighborhood`(p)
7: **if** $\Delta^* < 0$ **then** $f := f + \Delta^*$ **end if**
8: **end while**
9: $\mu :=$ `false`
10: $\Delta^* :=$ `explore_insertion_neighborhood`(p)
11: **if** $\Delta^* < 0$ **then**
12: $f := f + \Delta^*$
13: $\mu :=$ `true`
14: **end if**
15: **end while**
16: **return** f // possibly improved solution p is also returned

The pseudocode of the procedure `explore_swap_neighborhood` is given in Algorithm 5. The nested loops in Lines 2–10 implement Formulas (15) and (16). An improving move is represented by the triplet (t^*, k^*, l^*). If such a move has been detected, then the positions of the selected facilities are swapped in p_{t^*} (Line 12). This step is followed by updating the matrices used in the calculation of the move gain (Line 13). The following statement shows the efficiency of the procedure.

Algorithm 5 Swap neighborhood exploration procedure

`explore_swap_neighborhood`(p)
Input: Solution p.
Output: Best move gain Δ^* and solution p (improved if $\Delta^* < 0$).
1: $\Delta^* := 0$
2: **for** $t = 1, \ldots, m$ **do**
3: **for** each pair $s = p_t(k) \in H$ and $u = p_t(l) \in H$ such that $l > k$ and $L_s = L_u$ **do**
4: Compute $\Delta := \Delta(p, k, l)$ by (15) if $l > k + 1$ or by (16) if $l = k + 1$
5: **if** $\Delta < \Delta^*$ **then**
6: $\Delta^* := \Delta$
7: Set $t^* := t$, $k^* := k$, and $l^* := l$
8: **end if**
9: **end for**
10: **end for**
11: **if** $\Delta^* < 0$ **then**
12: Swap positions of facilities $p_{t^*}(k^*)$ and $p_{t^*}(l^*)$ in p_{t^*}
13: Update matrices C, C^-, C^+, X, D_{t^*}, and B_{t^*}
14: **end if**
15: **return** Δ^*

Proposition 3. *The computational complexity of the procedure* `explore_swap_neighborhood` *is* $O(mn^2)$.

Proof. Observe that loops 2–10 run in $O(mn^2)$ time. This is implied by the fact that the time complexity of the Δ calculation (Line 4) is $O(1)$. Other parts of the procedure have less complexity. The procedure performs $O(n^2)$ operations to update matrices D_{t^*} and B_{t^*}, $O(n)$ operations to update matrices C, C^-, and C^+, and $O(1)$ operations to update matrix X. □

We remark that the computational complexity of Algorithm 5 asymptotically matches the size of the neighborhood $\tilde{N}_1'$, which is $O(mn^2)$. Thus, the neighborhood exploration is performed in an efficient way. When the number of planning periods m is a constant, the running time of Algorithm 5 reduces to $O(n^2)$. As an alternative to the described procedure, one might use Equation (14) instead of (15) and (16) in Line 4 of Algorithm 5. However, the worst case complexity of such an implementation would be $O(mn^3)$.

To present our procedure for exploring the insertion neighborhood, we rewrite (12) as $\delta'(p, k, l) = \delta'(p, k, l + 1) + \delta_l^1$, where

$$\delta_l^1 = L_s(w_{tsv} - c_{tl}^-) + L_v(\alpha + \alpha' + c_{tk}^-) + G(v, t, x_{tv} + L_s), \tag{21}$$

$\alpha = \alpha_l$, and $\alpha' = \alpha_{l+1}$. Similarly, we rewrite (13) as $\delta'(p, k, l) = \delta'(p, k, l - 1) + \delta_l^2$, where

$$\delta_l^2 = L_s(w_{tsv} + c_{tl}^-) + L_v(\alpha' + \alpha - c_{tk}^-) + G(v, t, x_{tv} - L_s), \tag{22}$$

$\alpha' = \alpha_{l-1}$, and $\alpha = \alpha_l$.

The pseudocode of the procedure `explore_insertion_neighborhood` is shown in Algorithm 6. It can be seen that Lines 4–14 implement the first part of Proposition 1 (with (12)) and Lines 15–25 implement the second part of this proposition (with (13)). The sum $\Delta + \gamma$ stands for $\delta(p, k, l)$ as given by (6). The best move is stored using a triplet (t^*, k^*, l^*), where t^* is the selected period and k^* and l^* are the current and, respectively, the target position of the selected facility in permutation p_{t^*}. If an improving move is found, then the facility $p_{t^*}(k^*)$ is moved to position l^* (Line 29). Then, the matrices C, C^-, C^+, X, D_{t^*}, and B_{t^*} are updated in Line 30. The following result is similar to Proposition 3 and should be clear.

Proposition 4. *The computational complexity of the procedure* `explore_insertion_neighborhood` *is* $O(mn^2)$.

Algorithm 6 Insertion neighborhood exploration procedure

`explore_insertion_neighborhood`(p)
Input: Solution p.
Output: Best move gain Δ^* and solution p (improved if $\Delta^* < 0$).
$\Delta^* := 0$
for $t = 1, \ldots, m$ **do**
  for $k = 1, \ldots, n$ and $s = p_t(k)$ **do**
    Set $\Delta := 0$ and $\alpha := 0$ // α stands for α_l in (21)
    for $l = k - 1$ to 1 by -1, and $v = p_t(l)$ **do**
      $\alpha' := \alpha$ // α' stands for α_{l+1} in (21)
      $\alpha := \alpha + w_{tsv}$
      Compute δ_l^1 by (21) and $\gamma = G(s, t, x_{ts}(l))$
      $\Delta := \Delta + \delta_l^1$ // Δ stands for $\delta'(p, k, l)$ in (12)
      if $\Delta + \gamma < \Delta^*$ **then**
        $\Delta^* := \Delta + \gamma$
        Set $t^* := t$, $k^* := k$, and $l^* := l$
      end if
    end for
    Set $\Delta := 0$ and $\alpha := 0$ // α stands for α_l in (22)
    for $l = k + 1$ to n by 1, and $v = p_t(l)$ **do**
      $\alpha' := \alpha$ // α' stands for α_{l-1} in (22)
      $\alpha := \alpha + w_{tsv}$
      Compute δ_l^2 by (22) and $\gamma = G(s, t, x_{ts}(l))$
      $\Delta := \Delta + \delta_l^2$ // Δ stands for $\delta'(p, k, l)$ in (13)
      if $\Delta + \gamma < \Delta^*$ **then**
        $\Delta^* := \Delta + \gamma$
        Set $t^* := t$, $k^* := k$, and $l^* := l$
      end if
    end for
  end for
end for
if $\Delta^* < 0$ **then**
  Move facility $p_{t^*}(k^*)$ to position l^* in p_{t^*}
  Update matrices C, C^-, C^+, X, D_{t^*}, and B_{t^*}
end if
return Δ^*

We note that the size of the insertion neighborhood is $O(mn^2)$. It thus follows that the procedure takes $O(1)$ time per move. A variant of the procedure can be implemented by replacing Equations (12) and (13) with (7) and (9). However, such a replacement increases the time complexity to $O(mn^3)$.

4. Computational Results

In this section, we report on the results of computational tests to assess the performance of the developed variable neighborhood search algorithm for solving the DSRFLP. The effectiveness of the approach is evaluated by comparing the VNS against the simulated annealing algorithm proposed by Şahin et al. [2].

4.1. Experimental Setup

The VNS algorithm described in previous sections was coded in the C++ programming language. For comparison purposes, we also coded the simulated annealing algorithm of Şahin et al. [2]. These authors used the name SA-R to refer to this algorithm. We keep this

name. We ran SA-R with the parameter settings used in [2]. However, to be able to apply a time-based stopping criterion, we implemented SA-R as a multistart algorithm. Each time SA-R starts with a randomly generated solution. The experiments with VNS and SA-R were carried out on a PC with an Intel Core i5-9400F CPU running at 2.90 GHz.

Since research on the DSRFLP is in its infancy, there is no available benchmarks in the literature. Therefore, we performed our experiments on a set of randomly generated problem instances. The entries of the cost matrix (ψ_{su}) and the material flow matrices (ϕ_{tsu}), $t \in \{1, \ldots, m\}$, in these instances, are random numbers drawn uniformly between 1 and 5 and between 1 and 10, respectively. The costs associated with rearranging facilities are randomly and uniformly sampled from the interval $[250, 500]$ if $n \leqslant 100$ and interval $[1000, 2000]$ if $n > 100$. The length of each facility is a random integer number between 1 and 5. The generated instances have from 10 to 200 facilities. The number of planning periods is either 3 or 5. The dataset is publicly available (https://drive.google.com/file/d/1P36xdv4QpZxFmROhAa8Z2dOXU9zJOJKl/view?usp=sharing, accessed on 1 May 2022). Here, it is important to mention that, with the increase in the number of facilities, a more realistic layout scenario is to place facilities in a multi-row configuration. Our main reason for using single row layout problem instances with a large number of facilities (with $n > 100$) was to more comprehensively evaluate the performance of the described algorithms.

In our computational experiments, we ran both VNS and SA-R 10 times per instance. Maximum CPU time limits for a run of an algorithm were as follows: 180 s for $n \leqslant 20$, 600 s for $n = 30$, 1800 s for $40 \leqslant n \leqslant 60$, and 3600 s for $n \geqslant 70$. To measure the performance of the algorithms, we use the following statistics: the best objective function value from the 10 independent runs; the average objective function value for 10 runs; and the average time for reaching the best result.

4.2. Parameter Settings

In Section 2, we described two versions of our VNS implementation, VNS1 and VNS2. The first of them starts with a randomly generated permutation of facilities. Its parameters are ρ, $z_{\min}$, and θ (see Section 2). The second version (VNS2) starts with an initial solution produced by a VNS heuristic applied on an SRFLP instance. The run time of this heuristic is controlled through parameter β. Certainly, the parameters ρ, $z_{\min}$, and θ apply to VNS2, too.

To find good parameter settings for ρ, $z_{\min}$, and θ, we examined the performance of VNS on a training sample consisting of 10 DSRFLP instances with $n = 60, 70, 80, 90$, and 100 and $m = 3$ and 5. These instances were randomly generated as described in Section 4.1. Of course, the training sample is disjoint from the main dataset, which is reserved for the final testing stage. In each experiment, we ran several configurations of VNS. To evaluate them, we used the following formula

$$F_{\text{avdev}} = \sum (F - F_{\min})/10, \tag{23}$$

where F denotes the objective function value achieved by the tested configuration and $F_{\min}$ represents the minimum objective function value obtained by all configurations. The sum in (23) is taken over all instances in the sample. During the parameter tuning phase, we set the cutoff time of VNS to 300 s per instance.

In the stage of preliminary experimentation, we ran VNS1 and VNS2 using various combinations of the values of the parameters ρ, $z_{\min}$, and θ. We observed that the performance of VNS1 and VNS2 was not very sensitive to the choice of these parameters. Therefore, we relied on a simple parameter setting procedure for our algorithm. Based on early tests, we first identified a range of potential values for each parameter. Then, we allowed one parameter to take on different values from its range while keeping the other parameters fixed at reasonable values chosen on the basis of preliminary numerical experiments.

We started our numerical analysis by investigating the influence of the parameter ρ on the performance of the VNS algorithm. This parameter is used to define the maximum size of the

neighborhood in the search process. We ran VNS with $\rho \in \{0.1, 0.2, 0.3, 0.4, 0.5, 0.7, 1, 1.5, 2\}$. The results are plotted in Figure 4. We see that the best performance of VNS was for $\rho \leqslant 0.4$, with a slight edge to $\rho = 0.3$. Therefore, we fixed ρ at 0.3 for all further experiments with VNS. We then examined the following five values of the parameter $z_{\min}$: 1, 3, 5, 10, and 20. Figure 5 shows that the algorithm was fairly robust to the choice of $z_{\min}$. We decided to fix $z_{\min}$ at 3. Furthermore, we investigated the effect of the parameter θ on the performance of VNS. We ran the algorithm with $\theta = 1, 2, 3, 4, 5, 7, 10$, and 20. From Figure 6, we see that a bad choice is to set $\theta = 1$. The best values of θ appeared to be 2 and 5. The results were very similar between VNS configurations with other tested values of θ. Based on the obtained results, we fixed θ at 5.

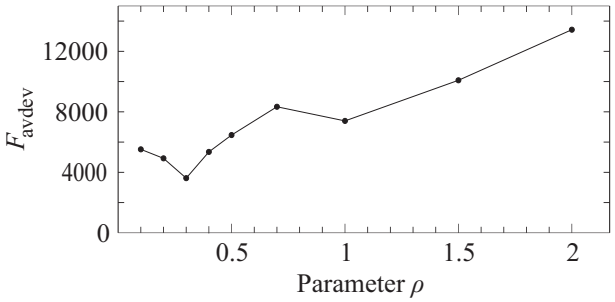

Figure 4. Average deviation F_{avdev} versus parameter ρ.

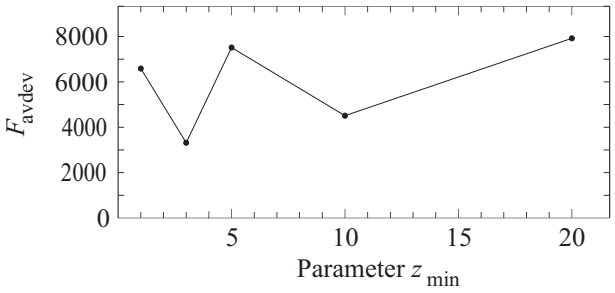

Figure 5. Average deviation F_{avdev} versus parameter $z_{\min}$.

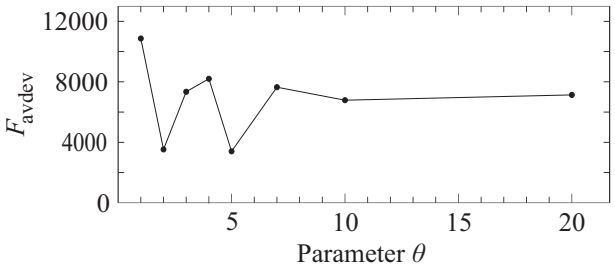

Figure 6. Average deviation F_{avdev} versus parameter θ.

As we alluded to at the beginning of this section, the VNS2 version of the algorithm makes use of the time share parameter β. This parameter represents the proportion of time allotted to a heuristic for providing an initial solution. We tested the values of β from 0 to 0.06 in increments of 0.01. In addition, we ran VNS2 with $\beta = 0.08, 0.1$, and 0.2. The results are displayed in Figure 7. We observe that the best performance of VNS2 was achieved for $\beta \in \{0.04, 0.05, 0.06\}$. We elected to set β to 0.04 for all further experiments with VNS2 reported in this paper.

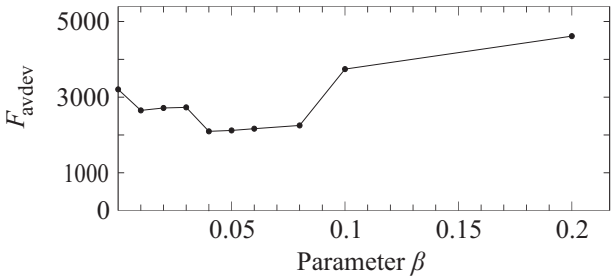

Figure 7. Average deviation F_{avdev} versus parameter β.

4.3. Computational Results for Smaller Sized Instances

We first report the computational experiments on a set of DSRFLP instances of size up to 100 facilities. Table 2 compares the best results achieved by VNS1, VNS2, and SA-R. Its first column contains the instance names. The first integer in the name gives the number of facilities and the second integer gives the number of planning periods. In the next columns, F^* is the objective function value of the best solution out of 10 runs. The average results of VNS1, VNS2, and SA-R are listed in Table 3. The two columns for each algorithm contain the average objective function value of 10 solutions, denoted as $\bar{F}$, and the average time (in seconds) taken to find the best solution in a run. The bottom row of Tables 2 and 3 shows the results averaged over all 20 problem instances. The best value of F^* (in Table 2) and $\bar{F}$ (in Table 3) for each instance is highlighted in boldface.

Table 2. Best results of VNS1, VNS2, and SA-R for smaller sized instances.

Instance	F^*		
	SA-R [2]	**VNS1**	**VNS2**
p-10-3	**11,705.04**	**11,705.04**	**11,705.04**
p-10-5	**36,542.40**	**36,542.40**	**36,542.40**
p-20-3	135,854.28	**135,836.72**	**135,836.72**
p-20-5	237,005.63	**235,702.67**	236,978.68
p-30-3	544,000.65	**543,651.77**	543,652.57
p-30-5	764,617.91	**760,389.18**	760,734.14
p-40-3	1,107,986.23	**1,105,774.53**	1,106,968.57
p-40-5	1,984,973.00	**1,977,976.69**	1,986,389.10
p-50-3	2,386,487.04	2,383,485.99	**2,383,192.05**
p-50-5	3,306,923.09	3,298,814.02	**3,298,351.85**
p-60-3	4,049,420.99	4,044,397.12	**4,043,248.10**
p-60-5	6,628,448.99	6,618,996.14	**6,618,257.45**
p-70-3	6,373,466.80	**6,367,620.87**	6,368,145.96
p-70-5	10,871,763.60	**10,846,635.18**	10,856,120.03
p-80-3	9,076,332.09	9,068,545.51	**9,067,154.61**
p-80-5	15,671,590.42	15,647,210.09	**15,644,661.53**
p-90-3	13,308,092.00	13,293,451.79	**13,289,264.47**
p-90-5	19,678,980.89	**19,642,957.24**	19,644,995.21
p-100-3	20,432,892.88	**20,406,423.73**	20,408,596.84
p-100-5	32,439,455.78	**32,376,234.74**	32,380,043.46
Average	7,452,326.99	7,440,117.57	7,441,041.94

By analyzing the results in Tables 2 and 3, we find that both VNS versions outperformed the SA-R algorithm. It can be observed that, for each instance of size greater than 10, VNS1 showed superior results to SA-R in terms of both performance measures, F^* and $\bar{F}$. Each algorithm found the best solution for the two smallest instances. However, SA-R produced better average solutions than VNS1 for these instances. We also see that VNS2 obtained better solutions than SA-R for 17 instances, matched the performance of SA-R for 2 instances, and failed in one case (for instance `p-40-5`). This observation is valid for each

table. The superiority of VNS over SA-R is also evidenced by the statistics presented in the last row of the tables, where we show the averaged results for each algorithm.

Table 3. Average results of VNS1, VNS2, and SA-R for smaller sized instances (the time is in seconds).

Instance	SA-R [2]		VNS1		VNS2	
	$\bar{F}$	Time	$\bar{F}$	Time	$\bar{F}$	Time
p-10-3	**11,705.04**	4	11,799.45	<1	**11,705.04**	<1
p-10-5	**36,542.40**	5	36,733.37	<1	**36,542.40**	<1
p-20-3	136,171.10	88	135,873.75	23	**135,836.72**	8
p-20-5	238,345.93	89	**236,178.93**	42	236,978.68	10
p-30-3	544,574.88	420	544,238.64	98	**544,018.00**	70
p-30-5	766,188.96	356	761,610.41	269	**760,892.22**	269
p-40-3	1,109,630.59	1161	1,108,546.91	627	**1,107,759.88**	922
p-40-5	1,986,043.95	840	**1,980,224.46**	837	1,986,508.74	1329
p-50-3	2,387,389.06	985	2,385,376.52	691	**2,383,939.81**	985
p-50-5	3,311,036.01	947	3,303,230.87	1262	**3,299,774.87**	1429
p-60-3	4,051,257.88	806	4,050,500.71	1244	**4,044,226.34**	1447
p-60-5	6,632,170.27	1089	6,622,791.04	1392	**6,621,988.34**	1437
p-70-3	6,374,969.44	1611	6,369,823.46	1646	**6,369,592.16**	2138
p-70-5	10,875,948.72	1806	**10,853,532.66**	2662	10,868,110.32	3093
p-80-3	9,079,287.72	1326	9,075,464.00	2553	**9,068,322.47**	2928
p-80-5	15,675,848.78	2682	15,656,867.55	3172	**15,650,223.41**	3223
p-90-3	13,316,487.34	1661	13,301,726.25	2830	**13,291,835.26**	2929
p-90-5	19,687,737.35	1394	19,659,048.67	3263	**19,658,507.91**	3330
p-100-3	20,437,133.73	1892	20,424,806.76	3004	**20,409,215.19**	2633
p-100-5	32,448,789.19	1563	32,398,327.96	3511	**32,394,880.52**	3320
Average	7,455,362.92	1036	7,445,835.12	1456	7,444,042.91	1575

Figure 8 depicts the boxplots of the performance of the tested algorithms on a subset of DSRFLP instances from Tables 2 and 3. The horizontal lines in each boxplot from bottom to top show the minimum, lower quartile, median, upper quartile, and maximum objective function values. As we can see in the figure, our VNS implementations were capable of delivering solutions of better quality than the SA-R algorithm.

Comparing VNS1 and VNS2, we find that VNS2 exhibits better performance than VNS1 in terms of $\bar{F}$, but is slightly worse in terms of F^*. To more rigorously assess the results, we apply the Wilcoxon signed-rank test. The comparison results are summarized in Table 4. The first column indicates which objective function values are compared: best solution values (F^* in Table 2) in the first row and average solution values ($\bar{F}$ in Table 3) in the second row. The next three columns show comparison results: #wins, #ties, and #losses count the number of instances on which VNS2 found a better, an equally good, or an inferior solution than VNS1. The p-values from the Wilcoxon test are estimated in the penultimate column. We use a standard significance level of 0.05 to judge whether a significant difference exists between the algorithms. The value "Yes" means that the average results of VNS2 are better than those of VNS1, while "No" means that there is no statistical difference between the two algorithms when the comparison is based on the F^* values.

Table 4. Comparison of VNS2 vs. VNS1 for smaller sized instances.

Objective Function Value	#wins	#ties	#losses	p-Value	Statistical Significance
Best	7	3	10	>0.2	No
Average	17	0	3	<0.025	Yes

The average running time taken by each algorithm to reach the last improvement in solution quality is shown in Table 3. We see that SA-R took less time to find the best solution in a run than VNS1 and VNS2. The running times of the latter two algorithms are quite comparable. Part (a) of Figure 9 shows the convergence speed of the tested algorithms

for problem instance p-100-5. For each algorithm, a plot of the objective function value of the best solution versus computational time is provided. The first point plotted for VNS2 represents the solution generated by a VNS heuristic for the SRFLP and improved by running LS once. From the figure, we see that SA-R stops improving the best solution earlier than VNS1 and VNS2.

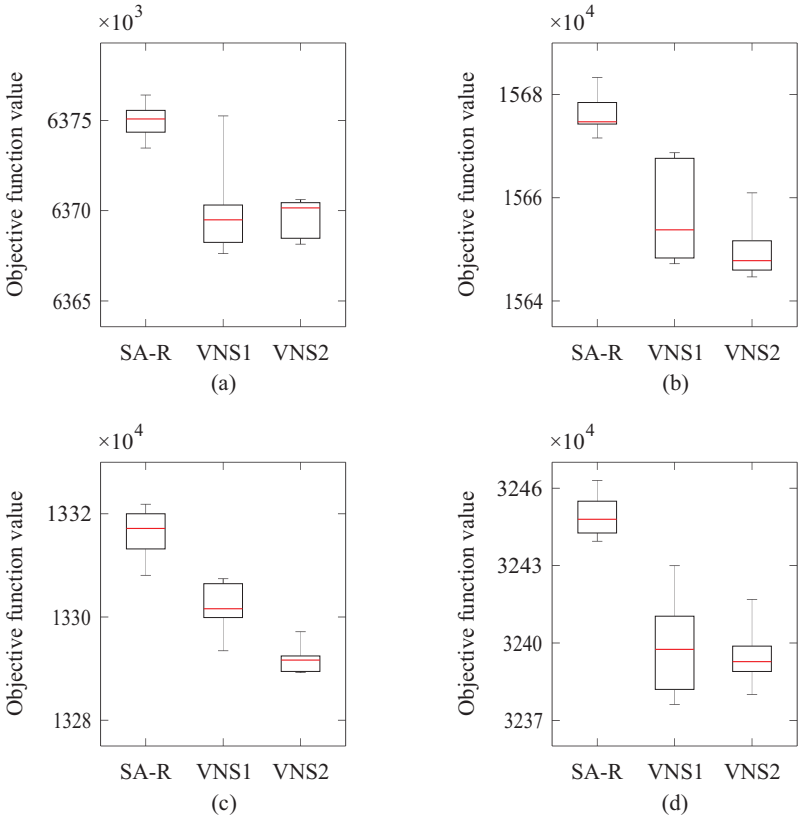

Figure 8. Objective function values achieved by SA-R, VNS1, and VNS2 on four DSRFLP instances with $n \leqslant 100$: (**a**) p-70-3; (**b**) p-80-5; (**c**) p-90-3; and (**d**) p-100-5.

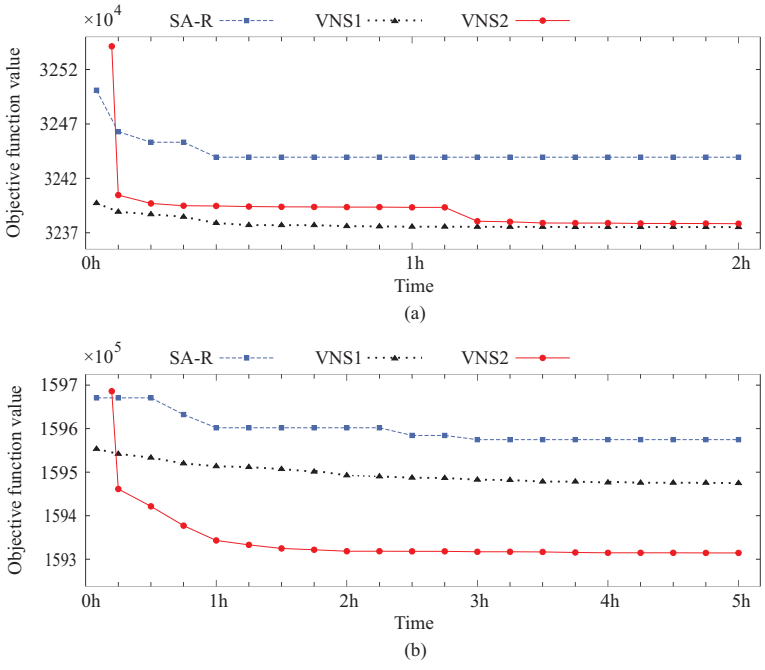

Figure 9. Convergence speed of SA-R, VNS1, and VNS2: (**a**) p-100-5; and (**b**) p-200-3.

4.4. Computational Results for Larger Sized Instances

Our second experiment aimed to assess the performance of algorithms on a set of DSRFLP instances with the number of facilities ranging from 110 to 200. The results are reported in Tables 5 and 6. They have the same structure as Tables 2 and 3. The findings from the experiment slightly differ from those discussed in the previous section. The differences are not only due to an increase in the size of problem instances. We used slightly different parameters to generate instances of size $n > 100$. The rearrangement costs for these instances are four times higher than in the case of instances of a size up to 100 (see Section 4.1).

Perhaps the main observation from Tables 5 and 6 is the overwhelming superiority of VNS2 over the other two algorithms. Basically, VNS2 dominated SA-R and VNS1 across all the 20 problem instances. By contrasting the penultimate column of Table 6 with the second and third columns of Table 5, we can see that the average result of VNS2 is better than the best SA-R (or VNS1) result for all instances except `p-130-3`. Comparing SA-R with VNS1, it can be noticed from the bottom rows of the tables that VNS1 found better "best" solutions (Table 5) and SA-R produced better solutions on the average (Table 6). The boxplots for the four DSRFLP instances with $n > 100$ are shown in Figure 10. They confirm the effectiveness of VNS2 in comparison with the other evaluated algorithms.

Table 5. Best results of VNS1, VNS2, and SA-R for larger-size instances.

Instance	F^*		
	SA-R [2]	**VNS1**	**VNS2**
p-110-3	24,426,599.17	24,412,874.74	**24,352,004.50**
p-110-5	44,394,528.07	44,394,371.91	**44,219,359.24**
p-120-3	32,482,176.39	32,492,680.33	**32,396,003.86**
p-120-5	51,521,726.72	51,584,284.77	**51,323,374.62**
p-130-3	43,120,283.15	43,112,625.52	**43,087,223.83**
p-130-5	67,282,522.47	67,318,731.00	**67,042,244.45**
p-140-3	52,320,015.69	52,301,770.34	**52,248,995.04**
p-140-5	80,707,509.29	80,793,212.05	**80,397,730.12**
p-150-3	66,670,525.25	66,685,317.12	**66,523,798.21**
p-150-5	110,264,451.42	110,259,943.20	**109,963,594.81**
p-160-3	75,165,009.15	75,141,439.40	**74,998,806.00**
p-160-5	143,801,397.84	143,699,497.42	**143,375,603.20**
p-170-3	90,472,044.23	90,527,250.24	**90,356,544.17**
p-170-5	140,374,941.98	140,448,285.02	**140,007,379.91**
p-180-3	100,099,923.84	100,029,906.48	**99,830,173.50**
p-180-5	187,124,398.15	186,995,144.87	**186,599,842.96**
p-190-3	129,442,600.58	129,392,623.33	**129,243,420.34**
p-190-5	223,475,308.22	223,303,403.18	**222,881,668.11**
p-200-3	159,575,676.04	159,458,326.81	**159,332,994.59**
p-200-5	256,724,704.71	256,210,522.64	**256,019,625.59**
Average	103,972,317.12	103,928,110.52	103,710,019.35

Table 7 summarizes comparison results between SA-R and VNS1. Regarding the average quality of solutions, the Wilcoxon signed-rank test demonstrated a statistically significant difference in favor of SA-R. However, there was no significant difference between the results of SA-R and VNS1 in the case of the best solutions.

In Table 6, we also report the average running time of the tested algorithms. We see that SA-R found the best solution in a run earlier than the VNS configurations. We notice that VNS2, and especially VNS1, obtained an improved solution in a situation where the time limit of 1 h on a run was close to expiring. In part (b) of Figure 9, we compare the convergence speed of the algorithms for the problem instance `p-200-3`. We increased the cutoff time for a run to 5 h. However, as we can see from the figure, after 3 h of execution, the improvement in solution quality was very marginal.

Table 6. Average results of VNS1, VNS2, and SA-R for larger-size instances (the time is in seconds).

Instance	SA-R [2]		VNS1		VNS2	
	$\bar{F}$	Time	$\bar{F}$	Time	$\bar{F}$	Time
p-110-3	24,438,826.09	1369	24,463,207.92	3531	**24,357,516.47**	3176
p-110-5	44,421,154.10	1626	44,472,395.59	3532	**44,247,884.94**	3278
p-120-3	32,498,608.51	1658	32,539,824.79	3525	**32,427,541.45**	3381
p-120-5	51,551,795.94	1235	51,656,799.26	3554	**51,361,807.61**	3394
p-130-3	43,137,728.60	1739	43,193,211.39	3543	**43,129,976.04**	3299
p-130-5	67,299,514.49	1496	67,404,238.70	3565	**67,102,663.67**	3379
p-140-3	52,348,807.27	1624	52,416,130.51	3509	**52,255,781.98**	3269
p-140-5	80,760,725.04	1928	80,875,690.25	3546	**80,425,261.19**	3509
p-150-3	66,695,736.62	1763	66,739,463.80	3546	**66,549,419.06**	3367
p-150-5	110,327,901.90	2040	110,406,239.53	3569	**110,076,167.87**	3346
p-160-3	75,188,758.41	2226	75,216,760.29	3518	**75,057,013.61**	3380
p-160-5	143,875,834.70	1459	143,886,630.77	3562	**143,565,705.90**	3396
p-170-3	90,533,129.91	2125	90,602,175.38	3547	**90,386,770.54**	3299
p-170-5	140,422,925.58	1657	140,499,276.62	3553	**140,057,222.24**	3219
p-180-3	100,119,474.17	2367	100,173,614.69	3570	**99,907,725.90**	3443
p-180-5	187,200,857.72	2420	187,149,490.63	3559	**186,717,534.74**	3425
p-190-3	129,490,670.18	1919	129,535,688.36	3525	**129,380,952.41**	3214
p-190-5	223,579,637.94	1709	223,546,537.36	3557	**223,003,189.23**	3456
p-200-3	159,609,830.84	1799	159,576,299.00	3561	**159,434,020.05**	3044
p-200-5	256,817,931.64	1475	256,646,600.68	3559	**256,200,019.90**	3214
Average	104,015,992.48	1782	104,050,013.78	3546	103,782,208.74	3324

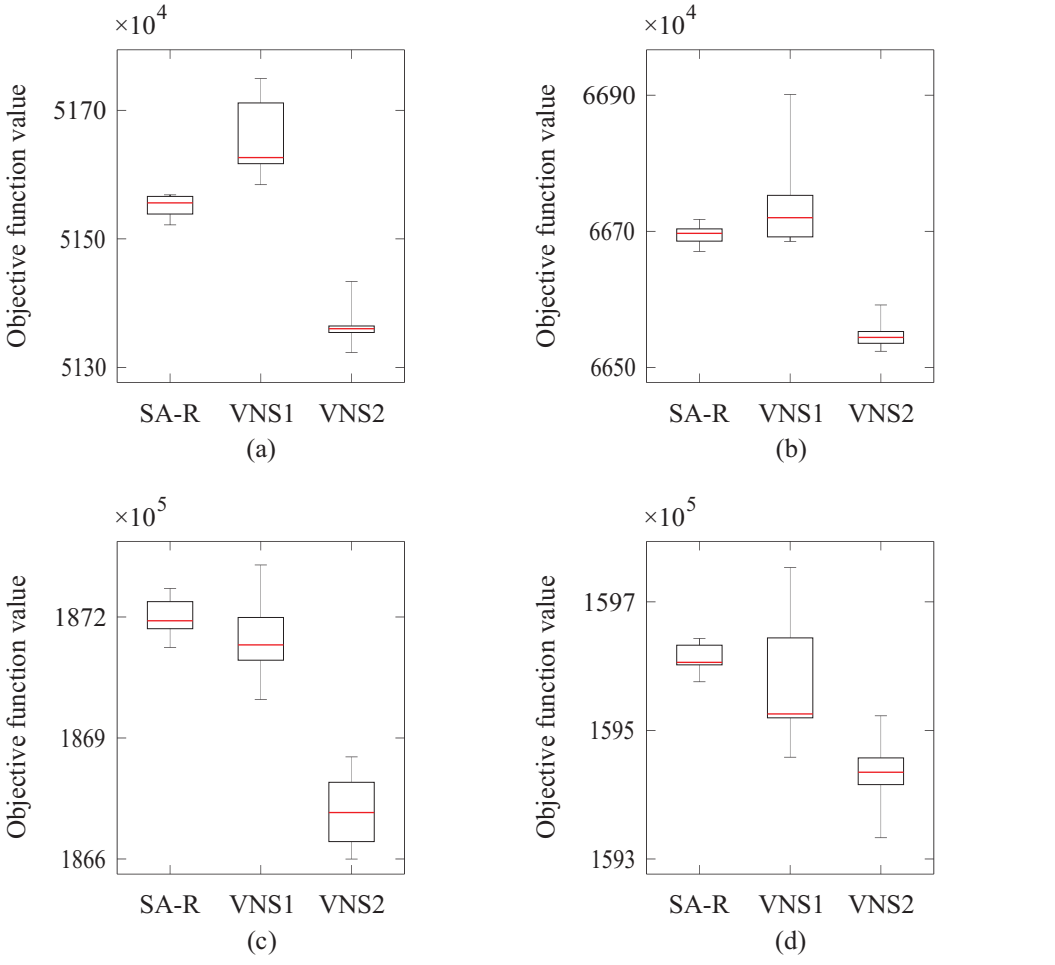

Figure 10. Objective function values achieved by SA-R, VNS1, and VNS2 on four DSRFLP instances with $n > 100$: (**a**) `p-120-5`; (**b**) `p-150-3`; (**c**) `p-180-5`; and (**d**) `p-200-3`.

Table 7. Comparison of SA-R vs. VNS1 for larger sized instances.

Objective Function Value	#wins	#losses	*p*-Value	Statistical Significance
Best	7	13	>0.2	No
Average	16	4	<0.025	Yes

We finalized this section with a couple of remarks concerning the performance of the tested algorithms. We experimentally compared our two algorithms (VNS1 and VNS2) for the DSRFLP. The results clearly indicate that VNS2 is our key algorithm in this study. We compared VNS2 with the simulated annealing algorithm (SA-R) from the literature. Experiments showed that the performance of VNS2 was superior to that of SA-R.

5. Analysis of Local Search Variants

To demonstrate the computational efficiency of our LS component of the algorithm, we experimentally compared it with alternative local search implementations. One idea was to abandon the use of the swap operator and base the search entirely on insertion moves. Other attempts were directed towards simplifying the gain calculation process by replacing the use of Propositions 1 and 2 with Equations (7), (9), and (14). Each of the resulting procedures, however, should not be considered as an independent algorithm. Basically, these procedures can be treated as modifications of VNS2. They are obtained by making small changes to the LS part of VNS2. We investigated alternative LS implementations in order to justify various design choices made in the construction of the VNS2 algorithm. To assess the performance of the variable neighborhood search algorithm with alternative LS approaches, we used the main VNS2 variant (described in Sections 2 and 3) as a reference method.

5.1. Usefulness of Swap Moves

We numerically analyzed the performance of a VNS2 version in which the LS procedure does not employ the swap neighborhood structure. We refer to this version as VNS2a. Basically, VNS2a is obtained from VNS2 by deleting Lines 4–8 in Algorithm 4.

To avoid unnecessarily long computations, we performed the comparison between VNS2 and VNS2a on a set of DSRFLP instances of Section 4.1 with $n \leqslant 100$. The comparison results are shown in Figure 11. On the x axis, $F_{\text{VNS2a}} - F_{\text{VNS2}}$ represents the difference in the F^* values between VNS2a and VNS2 in the bars labeled "Best", and the difference in the $\bar{F}$ values between VNS2a and VNS2 in the bars labeled as "Average". We provide results for problem instances of a size of at least 40. For smaller instances, the two versions of VNS2 either obtained the same solution or the difference between the objective function values was relatively small. We see in the figure that VNS2a found a better solution than VNS2 for `p-40-5`. The objective function value achieved by VNS2a for this instance is 1,982,841.80. This value, however, is bigger than that reported by VNS1 (see Table 2). We also observed that, for all problem instances of size greater than 40, VNS2 produced better solutions than VNS2a. The difference $F_{\text{VNS2a}} - F_{\text{VNS2}}$ averaged over all 20 instances was 8486.30 in the case of the F^* values and 11,322.90 in the case of the $\bar{F}$ values. From the figure, we can conclude that the swap neighborhood exploration procedure (Algorithm 5) is an important component of the approach. This helps significantly improve the quality of solutions produced by VNS.

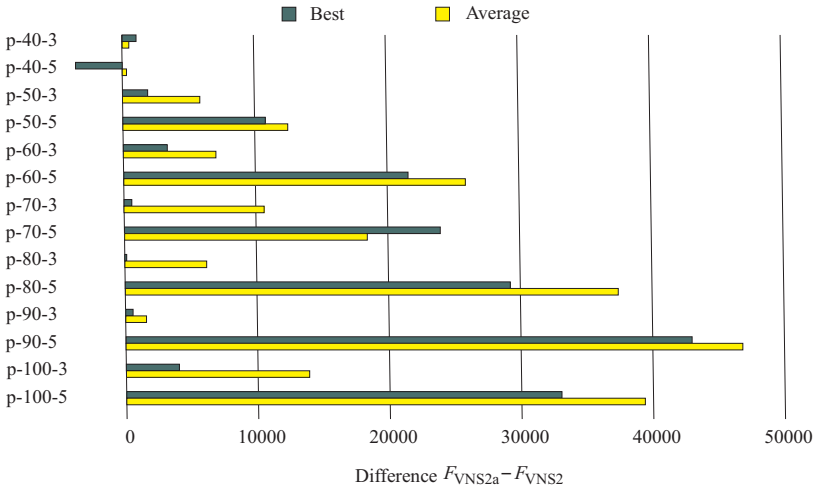

Figure 11. Difference in the solution values between VNS2a (that is, VNS2 with no swap operator) and the VNS2 algorithm.

5.2. *Benefit of Fast Neighborhood Exploration*

The purpose of this section is to show that our idea to calculate the move gain in constant time is fruitful. To this end, we consider the following variations of the VNS2 algorithm:

- VNS2b obtained from VNS2 by replacing the equations of Proposition 1 with Equations (7) and (9);
- VNS2c obtained from VNS2 by replacing Equation (15) with Equation (14);
- VNS2d obtained from VNS2 by replacing the equations of Propositions 1 and 2 with Equations (7), (9), and (14).

Since the VNS2 algorithm employs fast neighborhood exploration procedures, it was expected that the above-listed alternative VNS2 versions could not improve the results obtained by VNS2. Our experiment confirmed this expectation. As in Section 5.1, we ran the algorithm on problem instances of size $n \leqslant 100$. The results of solving these instances are reported in Figures 12 and 13. The differences shown in these figures for each VNS2 version were obtained similarly to those in Figure 11. We note that all versions of VNS2 managed to find the best solution for the first five instances in the dataset and therefore the results are only provided for `p-30-5` and all instances with $n \geqslant 40$. Figure 12 depicts the performance differences calculated for the F^* values and Figure 13 shows the performance differences obtained for the $\bar{F}$ values. As it is obvious from the figures, VNS2 produced better solutions than VNS2b, VNS2c, and VNS2d for all problem instances. The average values of $F_{VNS2b} - F_{VNS2}$, $F_{VNS2c} - F_{VNS2}$, and $F_{VNS2d} - F_{VNS2}$ in Figure 12 are 1146.54, 1305.63, and 1580.42, respectively, and those in Figure 13 are 1045.29, 1322.69, and 1686.44, respectively. As expected, VNS2d obtained worse solutions than VNS2b and VNS2c. In general, it can be concluded that the use of the proposed neighborhood exploration procedures has a large impact on improving the performance of the variable neighborhood search method for solving the DSRFLP.

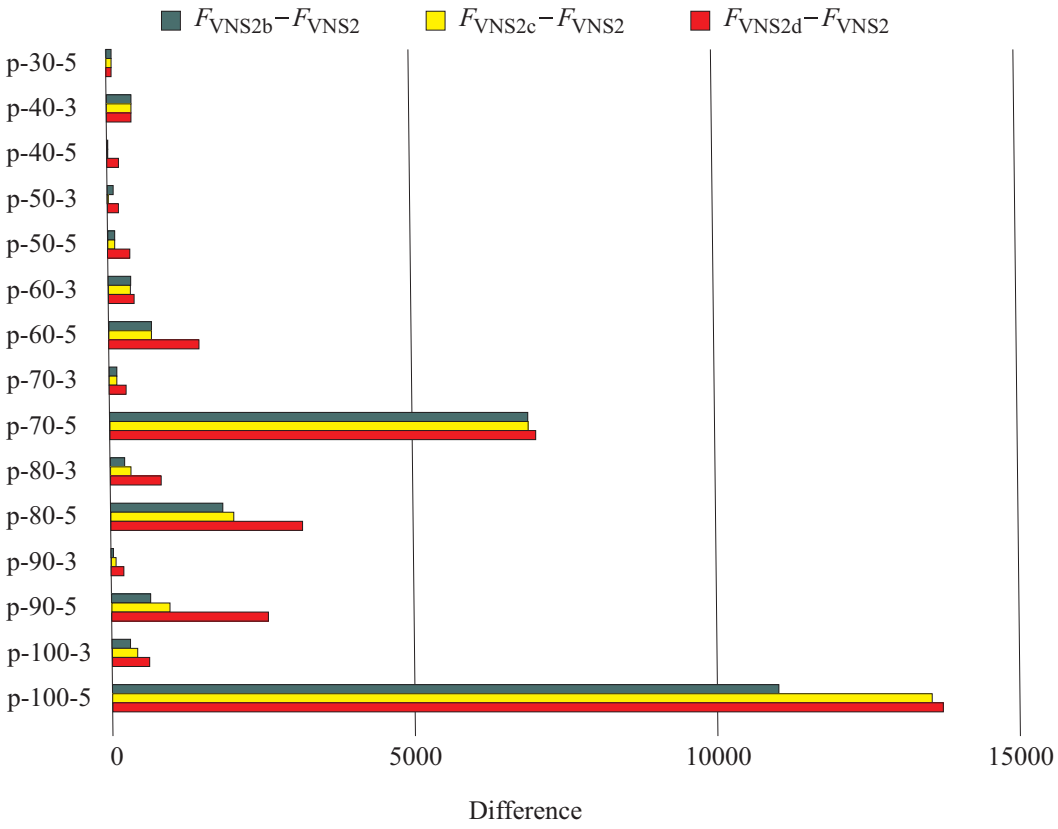

Figure 12. Difference in solution values between the VNS2 algorithm and VNS2 variations VNS2b, VNS2c, and VNS2d: the case of the F^* values.

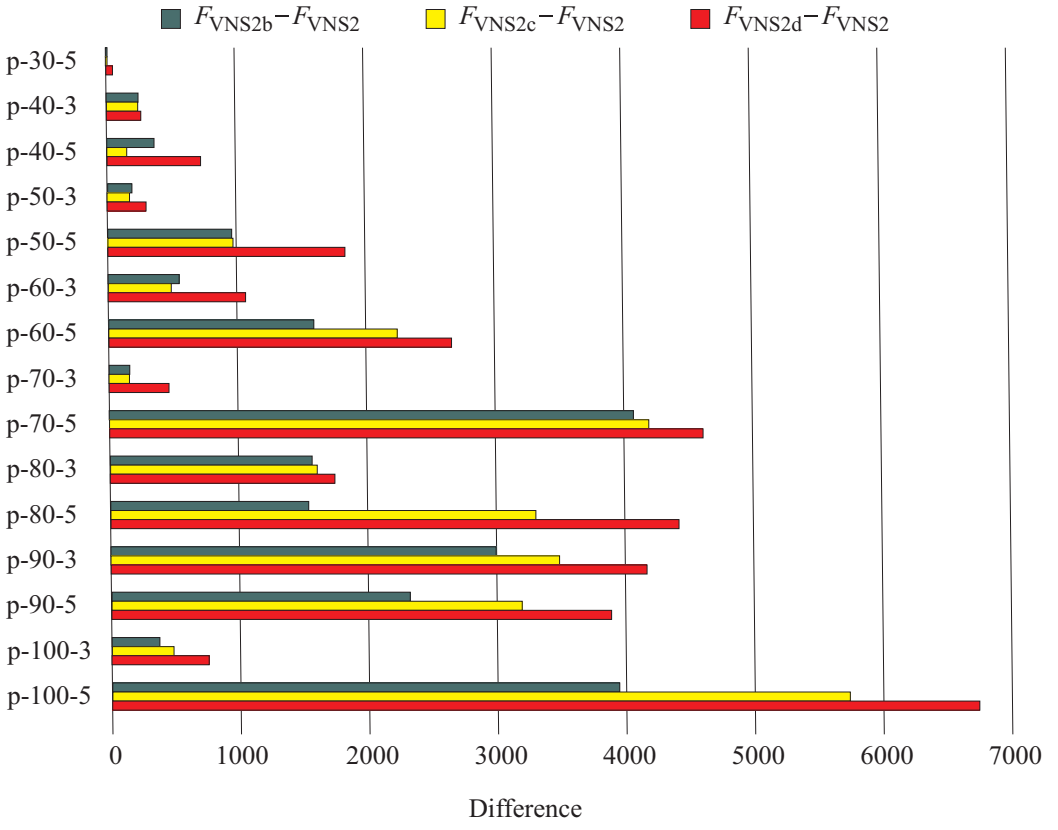

Figure 13. Difference in solution values between the VNS2 algorithm and VNS2 variations VNS2b, VNS2c, and VNS2d: the case of the $\bar{F}$ values.

6. Concluding Remarks

In this paper, we developed a variable neighborhood search algorithm for solving the dynamic single row facility layout problem. The effectiveness of this algorithm strongly depends on the quality of the adopted LS strategy. The main contribution of this work is an LS algorithm based on the fast neighborhood exploration procedures developed for swap and insertion neighborhood structures. Provided that the number of planning periods is a constant, these procedures have $O(n^2)$ time complexity. In one version of VNS, a starting solution is generated by performing the short run of a VNS procedure for solving the SRFLP. An instance of the latter is constructed from an instance of the DSRFLP. Another version of VNS starts with a randomly generated solution.

Both VNS versions were numerically compared against the SA approach of Şahin et al. [2], which is the state-of-the-art algorithm for the DSRFLP. Computational experiments were conducted on problem instances of size up to 200 facilities and 3 or 5 planning periods. The results indicate that VNS with a heuristically constructed initial solution outperformed the SA algorithm. The superiority of VNS over SA is more pronounced for DSRFLP instances of a size greater than 100.

Additional experiments were carried out to show the effectiveness of the proposed LS procedure. A conclusion was reached that it is advantageous to explore the swap neighborhood, and not to restrict LS to performing insertion operations only. Another conclusion is that our methods to calculate the move gain largely outperformed traditional techniques.

As a general conclusion, we may note that the DSRFLP is a very difficult combinatorial optimization problem. It is much harder than the SRFLP. We experience that, for large-size instances in our test suite, the best solutions obtained are likely not the best possible. To find improved solutions using VNS, a big amount of CPU time may be required. An obvious direction for further research is to develop new powerful metaheuristic-based algorithms for solving the DSRFLP. One idea is to combine SA and VNS into a single approach. This hybridization strategy was successfully applied to several permutation-based problems, for example, for solving the profile minimization problem [58]. Another promising idea seems to be developing evolutionary algorithms for the DSRFLP. Such algorithms may use the proposed LS procedure as a means for search intensification. In general, due to its complexity and simplicity in formulation, the DSRFLP can serve as a good candidate problem for the performance evaluation of newly introduced metaheuristic optimization methods.

Another possible area of future research is to design and implement metaheuristic algorithms for dynamic versions of some other facility layout problems. In particular, a research effort could be devoted to developing such algorithms for a dynamic formulation of the multi-row facility layout problem. Loop layout problems are another example for which dynamic models can be considered. Finally, the research can also be directed towards the development of new optimization techniques for dynamic parallel row ordering.

Author Contributions: Conceptualization, G.P.; methodology, G.P. and A.O.; software, G.P. and J.P.; validation, G.P., A.O. and J.P.; formal analysis, J.P.; investigation, G.P. and J.P.; resources, A.O.; writing—original draft preparation, G.P., A.O. and J.P.; writing—review and editing, G.P., A.O. and J.P.; supervision, G.P.; project administration, A.O. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

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

References

1. Keller, B.; Buscher, U. Single row layout models. *Eur. J. Oper. Res.* **2015**, *245*, 629–644. [CrossRef]
2. Şahin, R.; Niroomand, S.; Durmaz, E.D.; Molla-Alizadeh-Zavardehi, S. Mathematical formulation and hybrid meta-heuristic solution approaches for dynamic single row facility layout problem. *Ann. Oper. Res.* **2020**, *295*, 313–336. [CrossRef]
3. Amaral, A.R.S. An exact approach to the one-dimensional facility layout problem. *Oper. Res.* **2008**, *56*, 1026–1033. [CrossRef]
4. Amaral, A.R.S. A new lower bound for the single row facility layout problem. *Discret. Appl. Math.* **2009**, *157*, 183–190. [CrossRef]
5. Amaral, A.R.S.; Letchford, A.N. A polyhedral approach to the single row facility layout problem. *Math. Program.* **2013**, *141*, 453–477. [CrossRef]
6. Anjos, M.F.; Vannelli, A. Computing globally optimal solutions for single-row layout problems using semidefinite programming and cutting planes. *INFORMS J. Comput.* **2008**, *20*, 611–617. [CrossRef]
7. Hungerländer, P.; Rendl, F. A computational study and survey of methods for the single-row facility layout problem. *Comput. Optim. Appl.* **2013**, *55*, 1–20. [CrossRef]
8. Samarghandi, H.; Eshghi, K. An efficient tabu algorithm for the single row facility layout problem. *Eur. J. Oper. Res.* **2010**, *205*, 98–105. [CrossRef]
9. Kothari, R.; Ghosh, D. Tabu search for the single row facility layout problem using exhaustive 2-opt and insertion neighborhoods. *Eur. J. Oper. Res.* **2013**, *224*, 93–100. [CrossRef]
10. Datta, D.; Amaral, A.R.S.; Figueira, J.R. Single row facility layout problem using a permutation-based genetic algorithm. *Eur. J. Oper. Res.* **2011**, *213*, 388–394. [CrossRef]
11. Ozcelik, F. A hybrid genetic algorithm for the single row layout problem. *Int. J. Prod. Res.* **2012**, *50*, 5872–5886. [CrossRef]
12. Kothari, R.; Ghosh, D. An efficient genetic algorithm for single row facility layout. *Optim. Lett.* **2014**, *8*, 679–690. [CrossRef]
13. Kothari, R.; Ghosh, D. Insertion based Lin-Kernighan heuristic for single row facility layout. *Comput. Oper. Res.* **2013**, *40*, 129–136. [CrossRef]
14. Ou-Yang, C.; Utamima, A. Hybrid estimation of distribution algorithm for solving single row facility layout problem. *Comput. Ind. Eng.* **2013**, *66*, 95–103. [CrossRef]
15. Kothari, R.; Ghosh, D. A scatter search algorithm for the single row facility layout problem. *J. Heuristics* **2014**, *20*, 125–142. [CrossRef]
16. Palubeckis, G. Fast local search for single row facility layout. *Eur. J. Oper. Res.* **2015** *246*, 800–814. [CrossRef]
17. Rubio-Sánchez, M.; Gallego, M.; Gortázar, F.; Duarte, A. GRASP with path relinking for the single row facility layout problem. *Knowl. Based Syst.* **2016**, *106*, 1–13. [CrossRef]
18. Guan, J.; Lin, G. Hybridizing variable neighborhood search with ant colony optimization for solving the single row facility layout problem. *Eur. J. Oper. Res.* **2016**, *248*, 899–909. [CrossRef]
19. Palubeckis, G. Single row facility layout using multi-start simulated annealing. *Comput. Ind. Eng.* **2017** *103*, 1–16. [CrossRef]
20. Ning, X.; Li, P. A cross-entropy approach to the single row facility layout problem. *Int. J. Prod. Res.* **2018**, *56*, 3781–3794. [CrossRef]
21. Atta, S.; Sinha Mahapatra, P.R. Population-based improvement heuristic with local search for single-row facility layout problem. *Sādhanā* **2019**, *44*, 222. [CrossRef]
22. Cravo, G.L.; Amaral, A.R.S. A GRASP algorithm for solving large-scale single row facility layout problems. *Comput. Oper. Res.* **2019**, *106*, 49–61. [CrossRef]
23. Yeh, W.-C.; Lai, C.-M.; Ting, H.-Y.; Jiang, Y.; Huang, H.-P. Solving single row facility layout problem with simplified swarm optimization. In Proceedings of the 13th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD), Guilin, China, 29–31 July 2017; pp. 267–270. [CrossRef]
24. Krömer, P.; Platoš, J.; Snášel, V. Solving the single row facility layout problem by differential evolution. In Proceedings of the 2020 Genetic and Evolutionary Computation Conference (GECCO '20), Cancún, Mexico, 8–12 July 2020; ACM: New York, NY, USA, 2020. [CrossRef]
25. Di Bari, G.; Baioletti, M.; Santucci, V. An experimental evaluation of the algebraic differential evolution algorithm on the single row facility layout problem. In Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion (GECCO '20 Companion), Cancún, Mexico, 8–12 July 2020; ACM: New York, NY, USA, 2020. [CrossRef]
26. Sun, X.; Chou, P.; Koong, C.-S.; Wu, C.-C; Chen, L.-R. Optimizing 2-opt-based heuristics on GPU for solving the single-row facility layout problem. *Future Gener. Comput. Syst.* **2022**, *126*, 91–109. [CrossRef]
27. Anjos, M.F.; Vieira, M.V.C. Layout on a single row. In *Facility Layout. EURO Advanced Tutorials on Operational Research*; Springer: Cham, Switzerland, 2021. [CrossRef]
28. Kalita, Z.; Datta, D. A constrained single-row facility layout problem. *Int. J. Adv. Manuf. Technol.* **2018**, *98*, 2173–2184. [CrossRef]
29. Liu, S.; Zhang, Z.; Guan, C.; Zhu, L.; Zhang, M.; Guo, P. An improved fireworks algorithm for the constrained single-row facility layout problem. *Int. J. Prod. Res.* **2021**, *59*, 2309–2327. [CrossRef]
30. Keller, B. Construction heuristics for the single row layout problem with machine-spanning clearances. *INFOR Inf. Syst. Oper. Res.* **2019** *57*, 32–55. [CrossRef]
31. Amaral, A.R.S. A parallel ordering problem in facilities layout. *Comput. Oper. Res.* **2013** *40*, 2930–2939. [CrossRef]
32. Yang, X.; Cheng, W.; Smith, A.E.; Amaral, A.R.S. An improved model for the parallel row ordering problem. *J. Oper. Res. Soc.* **2020**, *71*, 475–490. [CrossRef]

33. Ahonen, H.; de Alvarenga, A.G.; Amaral, A.R.S. Simulated annealing and tabu search approaches for the corridor allocation problem. *Eur. J. Oper. Res.* **2014**, *232*, 221–233. [CrossRef]
34. Kalita, Z.; Datta, D.; Palubeckis, G. Bi-objective corridor allocation problem using a permutation-based genetic algorithm hybridized with a local search technique. *Soft Comput.* **2019**, *23*, 961–986. [CrossRef]
35. Zhang, Z.; Mao, L.; Guan, C.; Zhu, L.; Wang, Y. An improved scatter search algorithm for the corridor allocation problem considering corridor width. *Soft Comput.* **2020**, *24*, 461–481. [CrossRef]
36. Fischer, A.; Fischer, F.; Hungerländer, P. New exact approaches to row layout problems. *Math. Program. Comput.* **2019**, *11*, 703–754. [CrossRef]
37. Herrán, A.; Colmenar, J.M.; Duarte, A. An efficient variable neighborhood search for the space-free multi-row facility layout problem. *Eur. J. Oper. Res.* **2021**, *295*, 893–907. [CrossRef]
38. Gong, J.; Zhang, Z.; Liu, J.; Guan, C.; Liu, S. Hybrid algorithm of harmony search for dynamic parallel row ordering problem. *J. Manuf. Syst.* **2021**, *58*, 159–175. [CrossRef]
39. Guan, C.; Zhang, Z.; Zhu, L.; Liu, S. Mathematical formulation and a hybrid evolution algorithm for solving an extended row facility layout problem of a dynamic manufacturing system. *Robot. Comput.-Integr. Manuf.* **2022**, *78*, 102379. [CrossRef]
40. Rosenblatt, M.J. The dynamics of plant layout. *Manage. Sci.* **1986** *32*, 76–86. [CrossRef]
41. Balakrishnan, J.; Cheng, C.H. Genetic search and the dynamic layout problem. *Comput. Oper. Res.* **2000**, *27*, 587–593. [CrossRef]
42. Balakrishnan, J.; Cheng, C.H.; Conway D.G.; Lau C.M. A hybrid genetic algorithm for the dynamic plant layout problem. *Int. J. Prod. Econ.* **2003**, *86*, 107–120. [CrossRef]
43. McKendall, A.R.; Shang, J.; Kuppusamy S. Simulated annealing heuristics for the dynamic facility layout problem. *Comput. Oper. Res.* **2006**, *33*, 2431–2444. [CrossRef]
44. Baykasoglu, A.; Dereli, T.; Sabuncu I. An ant colony algorithm for solving budget constrained and unconstrained dynamic facility layout problems. *Omega* **2006**, *34*, 385–396. [CrossRef]
45. Zouein, P.P.; Kattan, S. An improved construction approach using ant colony optimization for solving the dynamic facility layout problem. *J. Oper. Res. Soc.* **2021**. [CrossRef]
46. McKendall, A.R.; Shang, J. Hybrid ant systems for the dynamic facility layout problem. *Comput. Oper. Res.* **2006**, *33*, 790–803. [CrossRef]
47. Şahin, R.; Türkbey, O. A new hybrid tabu-simulated annealing heuristic for the dynamic facility layout problem. *Int. J. Prod. Res.* **2009**, *47*, 6855–6873. [CrossRef]
48. McKendall, A.R.; Liu, W.-H. New tabu search heuristics for the dynamic facility layout problem. *Int. J. Prod. Res.* **2012**, *50*, 867–878. [CrossRef]
49. Hosseini-Nasab, H.; Emami, L. A hybrid particle swarm optimisation for dynamic facility layout problem. *Int. J. Prod. Res.* **2013**, *51*, 4325–4335. [CrossRef]
50. Turanoğlu, B.; Akkaya, G. A new hybrid heuristic algorithm based on bacterial foraging optimization for the dynamic facility layout problem. *Expert Syst. Appl.* **2018**, *98*, 93–104. [CrossRef]
51. Zhu, T.; Balakrishnan, J.; Cheng C.H. Recent advances in dynamic facility layout research. *INFOR Inf. Syst. Oper. Res.* **2018**, *56*, 428–456. [CrossRef]
52. Hosseini-Nasab, H.; Fereidouni, S.; Fatemi Ghomi, S.M.T.; Fakhrzad, M.B. Classification of facility layout problems: A review study. *Int. J. Adv. Manuf. Technol.* **2018**, *94*, 957–977. [CrossRef]
53. Mladenović, N.; Hansen, P. Variable neighborhood search. *Comput. Oper. Res.* **1997**, *24*, 1097–1100. [CrossRef]
54. Hansen, P.; Mladenović, N.; Moreno Pérez, J.A. Variable neighbourhood search: Methods and applications. *4OR* **2008**, *6*, 319–360. Erratumn in *Ann. Oper. Res.* **2010**, *175*, 367–407. [CrossRef]
55. Schiavinotto, T.; Stützle, T. A review of metrics on permutations for search landscape analysis. *Comput. Oper. Res.* **2007**, *34*, 3143–3153. [CrossRef]
56. Baioletti, M.; Milani, A.; Santucci, V. Variable neighborhood algebraic differential evolution: An application to the linear ordering problem with cumulative costs. *Inf. Sci.* **2020**, *507*, 37–52. [CrossRef]
57. Zaefferer, M.; Stork, J.; Bartz-Beielstein, T. Distance measures for permutations in combinatorial efficient global optimization. In *Lecture Notes in Computer Science, Proceedings of the Parallel Problem Solving from Nature—PPSN XIII, Ljubljana, Slovenia, 13–17 September 2014*; Bartz-Beielstein, T., Branke, J., Filipič, B., Smith, J., Eds.; Springer: Cham, Switzerland, 2014; Volume 8672, pp. 373–383. [CrossRef]
58. Palubeckis, G. A variable neighborhood search and simulated annealing hybrid for the profile minimization problem. *Comput. Oper. Res.* **2017**, *87*, 83–97. [CrossRef]