Applied Mathematics and Computation 486 (2025) 129040

Contents lists available at ScienceDirect

Applied Mathematics and Computation

journal homepage: www.elsevier.com/locate/amc

Full Length Article

A fast local search based memetic algorithm for the parallel row ordering problem

Gintaras Palubeckis

Faculty of Informatics, Kaunas University of Technology, Studentu 50, Kaunas, 51368, Lithuania

ARTICLE INFO

Keywords:
Combinatorial optimization
Local search
Memetic algorithm
Facility layout
Parallel row ordering problem

ABSTRACT

The parallel row ordering problem (PROP) is concerned with arranging two groups of facilities along two parallel lines with the goal of minimizing the sum of the flow cost-weighted distances between the pairs of facilities. As the main result of this paper, we show that the insertion neighborhood for the PROP can be explored in optimal time $\Theta(n^2)$ by providing an $O(n^2)$-time procedure for performing this task, where n is the number of facilities. As a case study, we incorporate this procedure in a memetic algorithm (MA) for solving the PROP. We report on numerical experiments that we conducted with MA on PROP instances with up to 500 facilities. The experimental results demonstrate that the MA is superior to the adaptive iterated local search algorithm and the parallel hyper heuristic method, which are state-of-the-art for the PROP. Remarkably, our algorithm improved best known solutions for six largest instances in the literature. We conjecture that the time complexity of exploring the interchange neighborhood for the PROP is $\Theta(n^2)$, exactly as in the case of insertion operation.

1. Introduction

One of the ways to increase manufacturing productivity is to optimize the placement of facilities (e.g., workstations or manufacturing cells) inside the plant. Optimization problems arising in this area are referred to as facility layout problems (FLPs). Typically, the objective function of this type of problems is minimizing the total material handling cost (MHC) [1,2]. The FLPs are widely acknowledged in the industry. As noted by Tompkins et al. [3], an effective facility layout can reduce the MHC by 10 to 30 percent annually. An important family of FLPs are row facility layout problems (RFLPs), where facilities are placed in one or more rows [4,5]. Perhaps the most studied RFLP is the single row facility layout problem (SRFLP). It involves arranging a given number of facilities in a single row with the goal of minimizing the sum of the flow cost-weighted distances between the pairs of facilities [6]. The solution space of the problem consists of all of the possible permutations of the facilities. One of the generalizations of the SRFLP is known as the parallel row ordering problem (PROP). This problem is concerned with placing facilities into two parallel straight lines (rows) and has the same objective as the SRFLP. It is assumed that the width of the corridor is negligible, and the distance between two facilities is taken as the x-distance between their centers. The PROP was introduced by Amaral [7].

To formulate the PROP, let n be the number of facilities and U_r, $r \in \{1,2\}$, be a set of n_r facilities to be placed in row r. Obviously, $n_r > 0$, $r = 1,2$, $n_1 + n_2 = n$, and $U_1 \cap U_2 = \emptyset$. We also denote by L_u the length of facility $u \in U = U_1 \cup U_2$ and by $W = (w_{uv})$ a symmetric matrix of size $n \times n$, with entry w_{uv} representing material transportation costs per distance unit between facility u and facility v. The solution space of the PROP is modeled by the set of all permutation pairs $p = (p_1, p_2)$, where $p_r = (p_r(1), \ldots, p_r(n_r))$,

E-mail address: gintaras.palubeckis@ktu.lt.

https://doi.org/10.1016/j.amc.2024.129040
Received 25 January 2024; Received in revised form 19 August 2024; Accepted 22 August 2024

Available online 28 August 2024

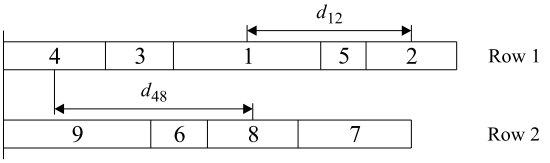

Fig. 1. Example of a solution of a PROP instance with 9 facilities.

Table 1
List of main notations.

Notation	Explanation
U	Set of facilities
$n = \|U\|$	Number of facilities
$r \in \{1,2\}$	Index for rows
U_r	Set of facilities to be placed in row r
n_r	Cardinality of the set U_r
s, u, v	Indices for facilities
L_u	Length of facility $u \in U$
$W = (w_{uv})$	Matrix of material transportation costs per distance unit
$\Pi^r(n_r)$	Set of all permutations on the set U_r
$p_r = (p_r(1), \ldots, p_r(n_r))$	Permutation in the set $\Pi^r(n_r)$
$p = (p_1, p_2)$	Solution to the PROP
$F(p_1, p_2)$	Objective function of the PROP
d_{uv}	Distance between facilities u and v
$x_u(p_r)$	X-coordinate of facility u in the permutation p_r
$\lambda^r = \{\lambda^r_1, \ldots, \lambda^r_{m_r}\}$	Ascendingly sorted array of lengths of facilities in U_r
m_r	Cardinality of the array λ^r ($m_r \leqslant n_r$)
ρ	Mapping $\rho(u) = i$, where $u \in U$ and $L_u = \lambda^r_i$ (assuming that $u \in U_r$)
$N(p)$	Insertion neighborhood of solution $p = (p_1, p_2)$
$\Delta(p_r, k, l)$	Gain of moving facility $p_r(k)$ to position l
$\delta(p_r, s, u)$	Gain of swapping facilities s and u, which are adjacent in p_r
P	Population of solutions maintained by the memetic algorithm

$r = 1, 2$, and $p_r(i) \in U_r$, $i \in I_r = \{1, \ldots, n_r\}$, is the ith facility in row r. For a facility $u \in U_r$, $r \in \{1,2\}$, we let $x_u(p_r)$ denote the x coordinate of the center of u in the arrangement defined by solution (p_1, p_2). Fig. 1 illustrates an instance of the PROP in which $n = 9$, $U_1 = \{1, \ldots, 5\}$, $U_2 = \{6, \ldots, 9\}$, $p_1 = (4, 3, 1, 5, 2)$, and $p_2 = (9, 6, 8, 7)$. Let $\Pi^r(n_r)$, $r \in \{1,2\}$, denote the set of all permutations on the set of facilities U_r. Mathematically, the PROP can be expressed as follows:

$$\min_{p_1 \in \Pi^1(n_1), p_2 \in \Pi^2(n_2)} F(p_1, p_2) = F_1(p_1) + F_2(p_2) + F_3(p_1, p_2), \tag{1}$$

$$F_r(p_r) = \sum_{i,j \in I_r, i<j} w_{p_r(i)p_r(j)} d_{p_r(i)p_r(j)}, r = 1, 2, \tag{2}$$

$$F_3(p_1, p_2) = \sum_{i \in I_1, j \in I_2} w_{p_1(i)p_2(j)} d_{p_1(i)p_2(j)}, \tag{3}$$

where

$$d_{p_r(i)p_q(j)} = d_{uv} = |x_u(p_r) - x_v(p_q)| \tag{4}$$

for $u = p_r(i)$, $v = p_q(j)$, and $r, q \in \{1,2\}$. If $r = q$ (and $i < j$ as in (2)), then

$$d_{p_r(i)p_r(j)} = (L_{p_r(i)} + L_{p_r(j)})/2 + \sum_{\substack{l=i+1 \\ \text{if } j>i+1}}^{j-1} L_{p_r(l)}. \tag{5}$$

Table 1 provides a list of key notations used in this study.

The PROP model (1)–(5) is suitable in layout design scenarios, where the set of facilities is split into groups. There are various factors that might be relevant when deciding whether to group facilities. They include machinery selection (e.g., the presence of heavy machines), physical and chemical characteristics of input materials, the nature of the product or services, the size and shape of the available space, maintenance, safety standards, and factory climate conditions (e.g., heat, light, sound, humidity, and ventilation). One specific application scenario is when facilities with similar functionality (e.g., chemical reactors) are placed in one row and the remaining facilities are placed in a parallel row. In another scenario, certain zoning constraints are imposed on the layout. For example, one or more zones can be established which are forbidden for placing certain facilities. A possible strategy is to split the set of facilities into two groups and arrange them in separate rows. One interesting scenario is when facilities exhibit significant

differences in their widths (dimensions along the y-axis). In this case, it may be advisable to assign wider facilities to one row and those of smaller width to another row. Such a strategy helps achieve a greater utilization of the layout area. An application scenario of the PROP model in the context of multi-floor building design was pointed out by Yang et al. [8]. A practical application of parallel row ordering in fermented food production was outlined by Gong et al. [9]. Potential applications of the PROP in the plant layout design phase motivate the development of algorithms for this recently formulated problem.

1.1. Related work

The PROP introduced by Amaral [7] is a relatively new problem in the field of facility layout. Nevertheless, there are several important papers in the literature addressing the PROP. Amaral [7] presented a mixed-integer programming (MIP) formulation of the PROP which was constructed by extending a MIP formulation of the SRFLP. The two formulations were compared with each other both theoretically and empirically. The largest PROP instances solved to optimality have 23 facilities. Yang et al. [8] proposed an improved MIP model for the PROP. It produced optimal solutions in significantly less computation time than the model of Amaral [7]. Moreover, the new formulation was effective in solving slightly larger PROP instances. Fischer et al. [10] presented a different mixed-integer linear programming (MILP) formulation for the PROP. Using this state-of-the-art formulation, the authors were able to solve to optimality instances with up to 25 facilities. Hungerländer and Anjos [11] formulated the PROP as a semidefinite optimization problem. They presented lower and upper bounds on the optimal value for large problem instances. Dahlbeck et al. [12] proposed a distance-based lower bounding MILP model for the PROP (and some other RFLPs). They provided experimental results for problem instances previously used in [10].

To deal with large-size problem instances, an effective solution is to apply metaheuristic techniques. In this line, Maadi et al. [13] proposed two approaches for the PROP, namely, a population based simulated annealing algorithm (PSA) and a genetic algorithm (GA). The authors provided the results of numerical experiments on a set of PROP benchmarks ranging in size from 30 to 70 facilities. Experiments have shown the advantages of the PSA over the GA implementation. Recently, Cravo and Amaral [14] presented an iterated local search (ILS) algorithm for solving the PROP. Their ILS, called AILS, is adaptive in a sense that it alternates between intensification and diversification phases using a self-regulating mechanism. The local search (LS) procedure of the algorithm is based on repeatedly applying swap moves to the current solution. The performance of LS is enhanced by means of a technique aimed at accelerating the computation of the objective function values. The developed AILS method found new best solutions for a large number of instances in the literature. Cravo and Amaral also presented computational results for large-scale problem instances ranging in size from 250 to 300 facilities. More recently, Liu et al. [15] proposed a parallel hyper heuristic algorithm based on reinforcement learning for solving the PROP. In this approach, the set of low-level heuristics consists of simulated annealing, tabu search, and variable neighborhood search algorithms. Reinforcement learning serves as a high-level heuristic strategy. The authors have experimentally shown the advantage of their hyper heuristic algorithm in comparison with low-level heuristics used as a stand-alone procedure. They compared the proposed algorithm with previous approaches on the benchmark PROP instances with $n \leqslant 70$. The results of computational experiments showed the competitiveness of their algorithm. Gong et al. [9] addressed the dynamic PROP in which the flow of material between facilities varies in different periods of time. To deal with this problem, they proposed an algorithm hybridizing harmony search and tabu search. The algorithm was tested both on the PROP benchmarks and dynamic PROP instances. The results of the algorithm were compared with those of GA and PSA algorithm proposed by Maadi et al. [13].

As already alluded to before, the PROP can be viewed as a generalization of the SRFLP. The latter has attracted much attention in recent years. Several exact methods have been developed for solving the SRFLP. They include semidefinite programming [16–18], cutting plane algorithm [19], mixed-integer linear programming [20,21], branch-and-cut [22], and a linear programming-based cutting plane method [23]. However, exact methods cannot solve large-scale SRFLP instances within a reasonable amount of time. A natural way to overcome this problem is to resort to metaheuristic methods. A variety of metaheuristic techniques for solving the SRFLP have been presented in the literature, such as genetic algorithms [24–26], tabu search [27], a cross-entropy method [28], variable neighborhood search (VNS) [29], a VNS and ant colony optimization hybrid [30], a path relinking enhanced greedy randomized adaptive search procedure (GRASP) [31], a scatter search method [32], multistart simulated annealing (SA) [33], GRASP [34], an improved fireworks algorithm [35], and a memetic algorithm with SA-based local optimizer [36]. Sun et al. [37] proposed a parallelization approach of 2-opt search algorithms for solving the SRFLP on graphic processing units. Recently, an interest towards a dynamic single row facility layout problem (DSRFLP) has arisen. Şahin et al. introduced this optimization problem in [38]. These authors developed two algorithms for the DSRFLP. The better one is based on the simulated annealing metaheuristic, and another one is a genetic algorithm. A VNS implementation for solving the DSRFLP was presented in [39].

There are a few other FLPs that bear some resemblance to the PROP. One of them is the corridor allocation problem (CAP) introduced by Amaral [40]. The difference with the PROP is that there is no requirement to split the set of facilities into two groups. To obtain optimal solutions for the CAP, Amaral [40] proposed a method based on a mixed-integer programming model. For large-scale CAP instances, different metaheuristic methods have been proposed, including tabu search [41], simulated annealing [41], genetic algorithms [42,43], a flower pollination algorithm [44], scatter search [45], a genetic algorithm and variable neighborhood search hybrid [46], a memetic algorithm [47], a clonal selection heuristic [48,49], a hybrid cloning selection algorithm [50], an algorithm hybridizing harmony search and tabu search [51], a parallel hyper heuristic based on reinforcement learning [15], and iterated local search [52]. In [53], a genetic programming hyper-heuristic approach for automatic generation of heuristics for the CAP was presented.

A significant amount of effort was also devoted to the development of algorithms for solving the double row layout problem (DRLP). There are two key differences between the PROP and the DRLP. First, as in the case of CAP, the latter does not require to provide a grouping of facilities. Second, in contrast to PROP, clearances between adjacent facilities are allowed. Chung and Tanchoco [54] developed the first MIP model for the DRLP. Subsequently, tighter MIP models were presented by Amaral [55,56] and Secchin and Amaral [57]. Later, Chae and Regan [58] and Amaral [59] introduced further improved MIP formulations. More recently, Amaral [60] proposed an enhanced MILP model for the problem and have shown its superiority over previous models. Several instances of size 20 were solved to optimality. A number of heuristic and metaheuristic-based techniques have also been proposed to deal with the DRLP. Murray et al. [61] presented an effective local search procedure featuring linear programming (LP). More recent approaches include a decomposition-based algorithm [62], a two-stage algorithm combining VNS and a sine-cosine heuristic [63], improvement heuristics with a LP mechanism [64], a discrete differential evolution (DDE) algorithm with an embedded two-phase method [65], and an improved DDE algorithm with LP [66]. Zuo et al. [67] studied the DRLP with shared clearances and proposed an approach based on hybridizing tabu search and heuristic rules for solving this problem. Wang et al. [68] addressed the dynamic DRLP and presented an algorithm combining an improved SA technique and mathematical programming.

The multi-row facility layout problem (MRFLP) [11] is another generalization of the SRFLP and PROP, in which facilities are arranged in two or more parallel rows. The MILP-based exact methods for the MRFLP were presented by Fischer et al. [10] and Anjos and Vieira [69]. A semidefinite programming approach to the MRFLP was proposed by Hungerländer and Anjos [11]. An integer linear programming model for the MRFLP with facilities of equal length was presented by Anjos et al. [70]. To handle large-scale instances, a number of heuristic algorithms were developed for the MRFLP: ant colony optimization [71], simulated annealing [71], genetic algorithm [72], particle swarm optimization [73], variable neighborhood search [74], a two stage optimization algorithm combining two mathematical optimization models [69], and a hybrid approach using genetic algorithm and tabu search [75]. There are several algorithms in the literature that employ a linear programming technique. Zuo et al. [76] proposed a three-stage approach in which a Monte Carlo heuristic is combined with a LP procedure. Wan et al. [77] provided a hybrid algorithm based on GRASP and LP. Guan et al. [78] developed a MILP model for the dynamic extended row FLP and presented a hybrid evolution algorithm to deal with this version of the MRFLP. Uribe et al. [79] addressed the multiple row equal facility layout problem (MREFLP). They proposed a GRASP with an improved local search for solving the MREFLP.

There are other important types of facility layouts that are similar to the RFLPs. One of them is a loop layout configuration where facilities are placed in a closed ring instead of assigning them to one or more parallel rows. Two versions of this problem, i.e., unidirectional [80] and bidirectional [81,82] loop layout design are studied in the literature. A more detailed discussion on facility layout problems can be found in the survey by Hosseini-Nasab et al. [5] and the recent book by Anjos and Vieira [83].

1.2. Contributions of this paper

One of the main components of various metaheuristic-based algorithms for combinatorial optimization problems is a local search procedure. When the feasible solution space of a problem consists of permutations (as in the case of the PROP), a good option is to develop an LS algorithm that is based on repeatedly applying an insertion operation. Typically, at each iteration, such an algorithm starts from the current solution (permutation) and explores its insertion neighborhood. The latter comprises all solutions that can be generated from the current permutation by performing an insertion move, which removes one facility from its original position in the permutation and inserts it in a new position. The time complexity of exploring the insertion neighborhood for the PROP, to our knowledge, is unknown in the literature. The primary motivation of this work is to settle this open problem by presenting a fast insertion neighborhood exploration procedure. This procedure is embedded into an LS scheme which can be incorporated as a search intensification mechanism into a variety of metaheuristic algorithms for the PROP. Our specific goal is to demonstrate that such algorithms can successfully compete with existing approaches. To this end, we have implemented a memetic algorithm (MA) that uses the developed LS procedure. One of the gaps in the literature on the PROP is a lack of fast local search techniques. This work fulfills this gap by presenting the MA for the PROP. Another gap is the need for algorithms capable of producing high-quality solutions for large PROP instances. Our algorithm aims to reduce this gap. It was tested on instances with several hundreds of facilities. Besides showing better LS performance, there are more differences between MA and the best existing algorithms, namely, AILS [14] and the parallel hyper heuristic algorithm (called QL-PHH in [15]). The MA is a population-based technique, whereas AILS is a single-solution algorithm and QL-PHH relies on using a set of low level heuristics. The MA operates by performing insertion moves, whereas both AILS and QL-PHH apply pairwise swaps of facilities. An attractive feature of the MA is that it has only two parameters, the population size and the generation size. In contrast, the AILS algorithm has four parameters and QL-PHH has more than 10 parameters.

The original contributions of the paper are summarized as follows.

- It is shown that, given a solution of the PROP, the insertion neighborhood of this solution can be explored in optimal time $\Theta(n^2)$, where n is the number of facilities. We develop a neighborhood exploration procedure that runs in $O(n^2)$ time, and thus is asymptotically best possible. This procedure is the key part of our LS algorithm.
- To illustrate the usefulness of the proposed LS technique, we present a case study where this technique is used to develop a memetic algorithm for the PROP.
- We report on numerical experiments that we conducted with MA on PROP instances with up to 500 facilities. Their results demonstrate the superior performance of MA compared to state-of-the-art approaches: adaptive ILS algorithm of Cravo and

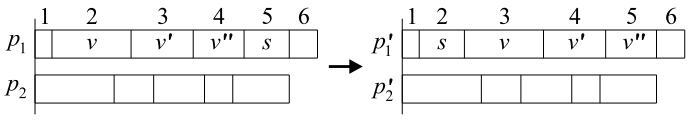

Fig. 2. Moving facility s to position $l = 2$.

Amaral [14] and parallel hyper heuristic algorithm of Liu et al. [15]. In particular, we improve 6 best known solutions for large-scale PROP instances in the literature.

The structure of this paper is the following. Section 2 presents the insertion-based local search technique for the PROP. Section 3 describes our memetic algorithm for this problem. Section 4 gives our numerical results and provides performance comparisons with benchmark algorithms. Finally, Section 5 summarizes our contributions and discusses the future research directions.

2. Insertion-based local search

This section describes a local search algorithm that explores the solution space by repeatedly applying the insertion operation until a local minimum is reached. Let the current solution during this process be denoted as a pair $p = (p_1, p_2)$, $p_r \in \Pi'(n_r)$, $r = 1, 2$, and its insertion neighborhood as $N(p)$. This neighborhood is defined as a set of all solutions obtainable from (p_1, p_2) by shifting a facility either to the left or right by one or more positions in the corresponding permutation. Given a permutation $p_r \in \Pi'(n_r)$, $r \in \{1, 2\}$, let us denote by p'_r the permutation obtained from p_r by inserting facility $s = p_r(k)$, $k \in \{1, \dots, n_r\}$, at position $l \in \{1, \dots, n_r\}$, $l \neq k$. An example of such a move for p_1 is visualized in Fig. 2, where $k = 5$ and $l = 2$. The difference between objective function values of the new solution, obtained by moving facility $s = p_r(k)$ to position l, and the previous solution (p_1, p_2) is $\Delta(p_r, k, l) = F(p'_1, p_2) - F(p_1, p_2)$ if $r = 1$ and $\Delta(p_r, k, l) = F(p_1, p'_2) - F(p_1, p_2)$ if $r = 2$. Suppose that facility s is moved to the left. It is convenient to decompose this move into a series of steps, each exchanging the positions of facility s and its current left-side neighbor. In this operation, first, s is interchanged with the facility $p_r(k-1)$. Then, the exchanging steps are executed for facility pairs $(s, p_r(k-2)), (s, p_r(k-3)), \dots, (s, p_r(l))$. Let us assume that the current left neighbor of s is facility u. Swapping positions of facilities s and u may change the value of the objective function. We denote the change by $\delta(p_r, s, u)$. Then, $\Delta(p_r, k, l)$ can be written as a sum of $\delta(p_r, s, p_r(j))$, $j = k-1, k-2, \dots, l$. In Fig. 2, $\Delta(p_1, 5, 2) = \delta(p_1, s, v'') + \delta(p_1, s, v') + \delta(p_1, s, v)$. Similarly, if $l > k$, then $\Delta(p_r, k, l)$ is expressed as a sum of $l - k$ terms $\delta(p_r, s, p_r(j))$, $j = k+1, k+2, \dots, l$. Thus, the move gains $\Delta(p_r, k, l)$ can be computed iteratively by successively calculating the δ values:

$$\Delta(p_r, k, l) = \begin{cases} \Delta(p_r, k, l+1) + \delta(p_r, s, u) & \text{if } l < k \\ \Delta(p_r, k, l-1) + \delta(p_r, s, u) & \text{if } l > k, \end{cases} \tag{6}$$

where $u = p_r(l)$ and $\Delta(p_r, k, k) = 0$.

Since the facilities are placed in two rows it is convenient to split $\Delta(p_r, k, l)$ into two parts:

$$\Delta(p_r, k, l) = \Delta^1(p_r, k, l) + \Delta^2(p_r, k, l),$$

where $\Delta^1(p_r, k, l) = F_r(p'_r) - F_r(p_r)$ and $\Delta^2(p_r, k, l) = F_3(p'_1, p_2) - F_3(p_1, p_2)$ if $r = 1$ and $\Delta^2(p_r, k, l) = F_3(p_1, p'_2) - F_3(p_1, p_2)$ if $r = 2$. Similarly, $\delta(p_r, s, u)$ is split into $\delta^1(p_r, s, u)$ and $\delta^2(p_r, s, u)$ representing the change in the value of the function $F_1(p_1) + F_2(p_2)$ and $F_3(p_1, p_2)$, respectively. For $l < k$ and $u = p_r(l)$, we can write

$$\Delta^i(p_r, k, l) = \Delta^i(p_r, k, l+1) + \delta^i(p_r, s, u), i = 1, 2. \tag{7}$$

Similarly, for $l > k$ and $u = p_r(l)$, we have

$$\Delta^i(p_r, k, l) = \Delta^i(p_r, k, l-1) + \delta^i(p_r, s, u), i = 1, 2. \tag{8}$$

In order to be able to apply Equations (7) and (8), we need expressions for $\delta^i(p_r, s, u)$, $i = 1, 2$. First, our focus is on $\delta^2(p_r, s, u)$. To derive the required formulas, we need additional notations. Let $\lambda^r = \{\lambda^r_1, \dots, \lambda^r_{m_r}\}$, $r \in \{1, 2\}$, be the array of lengths of facilities in U_r. It is obtained simply by removing duplicates from the array $\{L_u \mid u \in U_r\}$ (thus $m_r \leqslant n_r$). We assume that λ^r, $r \in \{1, 2\}$, is sorted in ascending order. We also define a mapping ρ on U as follows: $\rho(u) = i$, where i is such that $L_u = \lambda^r_i$, provided $u \in U_r$, $r \in \{1, 2\}$. Continuing to assume that $u \in U_r$, we consider the set $Z(u) = \{i = 1, \dots, n_{3-r} \mid x_v(p_{3-r}) \leqslant x_u(p_r), v = p_{3-r}(i)\}$. We define

$$z_u = \begin{cases} \max\{i \mid i \in Z(u)\} & \text{if } Z(u) \neq \emptyset \\ 0 & \text{otherwise.} \end{cases}$$

For example, in Fig. 3, $z_u = 2$ and $z_s = 6$. Based on z_u, we can define the sums $g^{\text{left}}_u = \sum_{i=1}^{z_u} w_{up_{3-r}(i)}$ and $g^{\text{right}}_u = \sum_{i=z_u+1}^{n_{3-r}} w_{up_{3-r}(i)}$ (we assume that $g^{\text{left}}_u = 0$ if $z_u = 0$ and $g^{\text{right}}_u = 0$ if $z_u = n_{3-r}$). Since $g^{\text{tot}}_u = \sum_{v \in U_{3-r}} w_{uv}$ is constant in the solution space of the problem, we can write $g^{\text{right}}_u = g^{\text{tot}}_u - g^{\text{left}}_u$. In Fig. 3, $g^{\text{left}}_s = w_{sp_2(1)} + \dots + w_{sp_2(6)}$ and $g^{\text{right}}_s = w_{sp_2(7)} + w_{sp_2(8)}$. To compute $\delta^2(p_r, s, u)$, we also use the set $H_{3-r,su} = \{p_{3-r}(i) \mid i = 1, \dots, z_s, x_v(p_{3-r}) \geqslant x_s(p_r) - L_u, v = p_{3-r}(i)\}$ in the case where s is interchanged with its left neighbor u (Fig. 3) and the set $H'_{3-r,su} = \{p_{3-r}(i) \mid i = z_s + 1, \dots, n_{3-r}, x_v(p_{3-r}) \leqslant x_s(p_r) + L_u, v = p_{3-r}(i)\}$ in the case where s is interchanged with

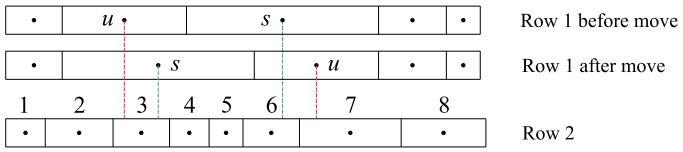

Fig. 3. Moving facility s to the left by one step.

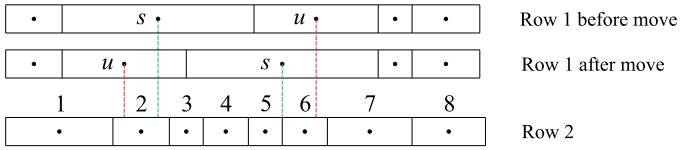

Fig. 4. Moving facility s to the right by one step.

its right neighbor u (Fig. 4). In Fig. 3, $H_{2,su} = \{p_2(4), p_2(5), p_2(6)\}$ and, in Fig. 4, $H'_{2,su} = \{p_2(3), p_2(4), p_2(5)\}$. Assume $u \in U_r$ as before, and consider $i \in \{1, \dots, m_r\}$. Given such u and i, we define the set $V_{ui} = \{v \in U_{3-r} \mid x_u(p_r) < x_v(p_{3-r}) < x_u(p_r) + \lambda_i^r\}$ and the sums $A_{ui}^{\text{right}} = \sum_{v \in V_{ui}} w_{uv}$ and $B_{ui}^{\text{right}} = \sum_{v \in V_{ui}} x_v(p_{3-r}) w_{uv}$. In a similar way, we define $V'_{ui} = \{v \in U_{3-r} \mid x_u(p_r) - \lambda_i^r < x_v(p_{3-r}) \leqslant x_u(p_r)\}$ and the sums $A_{ui}^{\text{left}} = \sum_{v \in V'_{ui}} w_{uv}$ and $B_{ui}^{\text{left}} = \sum_{v \in V'_{ui}} x_v(p_{3-r}) w_{uv}$. For $u \in U_1$ and $i = \rho(s) = 5$, in Fig. 3, $V_{u5} = \{p_2(3), p_2(4), p_2(5), p_2(6)\}$, and in Fig. 4, $V'_{u5} = \{p_2(2), p_2(3), p_2(4), p_2(5), p_2(6)\}$.

Now, we are ready to present an expression for $\delta^2(p_r, s, u)$.

Proposition 1. *For a facility $s \in U_r$, $r \in \{1, 2\}$, and its left neighbor u,*

$$\delta^2(p_r, s, u) = \delta_1(s) + \delta_2(s) - \delta_3(s) + \delta_4(u) + \delta_5(u) - \delta_6(u), \tag{9}$$

where

$$\delta_1(s) = g_s^{\text{right}} L_u, \tag{10}$$

$$\delta_2(s) = \sum_{v \in H_{3-r,su}} (L_u - 2(x_s(p_r) - x_v(p_{3-r}))) w_{sv}, \tag{11}$$

$$\delta_3(s) = (g_s^{\text{left}} - \sum_{v \in H_{3-r,su}} w_{sv}) L_u, \tag{12}$$

$$\delta_4(u) = g_u^{\text{left}} L_s, \tag{13}$$

$$\delta_5(u) = (L_s + 2x_u(p_r)) A_{u,\rho(s)}^{\text{right}} - 2B_{u,\rho(s)}^{\text{right}}, \tag{14}$$

$$\delta_6(u) = (g_u^{\text{right}} - A_{u,\rho(s)}^{\text{right}}) L_s. \tag{15}$$

For a facility $s \in U_r$, $r \in \{1, 2\}$, and its right neighbor u,

$$\delta^2(p_r, s, u) = \delta'_1(s) + \delta'_2(s) - \delta'_3(s) + \delta'_4(u) + \delta'_5(u) - \delta'_6(u), \tag{16}$$

where

$$\delta'_1(s) = g_s^{\text{left}} L_u, \tag{17}$$

$$\delta'_2(s) = \sum_{v \in H'_{3-r,su}} (L_u - 2(x_v(p_{3-r}) - x_s(p_r))) w_{sv}, \tag{18}$$

$$\delta'_3(s) = (g_s^{\text{right}} - \sum_{v \in H'_{3-r,su}} w_{sv}) L_u, \tag{19}$$

$$\delta'_4(u) = g_u^{\text{right}} L_s, \tag{20}$$

$$\delta'_5(u) = (L_s - 2x_u(p_r)) A_{u,\rho(s)}^{\text{left}} + 2B_{u,\rho(s)}^{\text{left}}, \tag{21}$$

$$\delta'_6(u) = (g_u^{\text{left}} - A_{u,\rho(s)}^{\text{left}}) L_s. \tag{22}$$

Proof. Denote by p'_r the permutation produced from p_r by exchanging facilities s and u. For a facility $v \in U_{3-r}$, let

$$\tilde{\delta}(p_r, p'_r, t, v) = w_{tv}(|x_t(p'_r) - x_v(p_{3-r})| - |x_t(p_r) - x_v(p_{3-r})|), \tag{23}$$

where t is either s or u. We extend this definition for a subset U' of U_{3-r} by letting $\tilde{\delta}(p_r, p'_r, t, U') = \sum_{v\in U'} \tilde{\delta}(p_r, p'_r, t, v)$. Assume that the set U_{3-r} is partitioned into mutually disjoint subsets $U^1_{3-r}, \dots, U^\eta_{3-r}$ with respect to facility s and into mutually disjoint subsets $\hat{U}^1_{3-r}, \dots, \hat{U}^\mu_{3-r}$ with respect to facility u. Since the facilities in $U \setminus \{s, u\}$ are not affected by the move, we can write

$$\delta^2(p_r, s, u) = \sum_{i=1}^{\eta} \tilde{\delta}(p_r, p'_r, s, U^i_{3-r}) + \sum_{i=1}^{\mu} \tilde{\delta}(p_r, p'_r, u, \hat{U}^i_{3-r}). \tag{24}$$

We first consider the case where $s \in U_r$ and u is the left neighbor of s. We partition U_{3-r} into $U^1_{3-r} = \{p_{3-r}(i) \mid i = z_s + 1, \dots, n_{3-r}\}$, $U^2_{3-r} = H_{3-r,su}$, and $U^3_{3-r} = U_{3-r} \setminus (U^1_{3-r} \cup U^2_{3-r})$ (thus, we have that $\eta = 3$). In Fig. 3, $U^1_{3-r} = \{p_2(7), p_2(8)\}$, $U^2_{3-r} = \{p_2(4), p_2(5), p_2(6)\}$, and $U^3_{3-r} = \{p_2(1), p_2(2), p_2(3)\}$. Using (23), we find that $\tilde{\delta}(p_r, p'_r, s, U^1_{3-r}) = \sum_{v\in U^1_{3-r}} w_{sv}(x_s(p_r) - x_s(p'_r)) = g^{\text{right}}_s L_u = \delta_1(s)$. Similarly, $\tilde{\delta}(p_r, p'_r, s, U^2_{3-r}) = \sum_{v\in U^2_{3-r}} w_{sv}((x_v(p_{3-r}) - x_s(p'_r)) - (x_s(p_r) - x_v(p_{3-r})))$. Since $x_s(p'_r) = x_s(p_r) - L_u$ it follows that $\tilde{\delta}(p_r, p'_r, s, U^2_{3-r}) = \sum_{v\in U^2_{3-r}} w_{sv}(2x_v(p_{3-r}) - 2x_s(p_r) + L_u) = \delta_2(s)$. For the set U^3_{3-r}, we have $\tilde{\delta}(p_r, p'_r, s, U^3_{3-r}) = \sum_{v\in U^3_{3-r}} w_{sv}(x_s(p'_r) - x_s(p_r)) = -L_u \sum_{v\in U^3_{3-r}} w_{sv} = -L_u(\sum_{v\in U_{3-r}\setminus U^1_{3-r}} w_{sv} - \sum_{v\in U^2_{3-r}} w_{sv}) = -L_u(g^{\text{left}}_s - \sum_{v\in H_{3-r,su}} w_{sv}) = -\delta_3(s)$. Considering the second sum in (24), we partition U_{3-r} into $\mu = 3$ disjoint subsets $\hat{U}^1_{3-r} = \{p_{3-r}(i) \mid i = 1, \dots, z_u\}$, $\hat{U}^2_{3-r} = \{v \in U_{3-r} \mid x_u(p_r) < x_v(p_{3-r}) < x_u(p_r) + L_s\}$, and $\hat{U}^3_{3-r} = U_{3-r} \setminus (\hat{U}^1_{3-r} \cup \hat{U}^2_{3-r})$. In Fig. 3, $\hat{U}^1_{3-r} = \{p_2(1), p_2(2)\}$, $\hat{U}^2_{3-r} = \{p_2(3), p_2(4), p_2(5), p_2(6)\}$, and $\hat{U}^3_{3-r} = \{p_2(7), p_2(8)\}$. To derive expressions for the $\tilde{\delta}$ terms in the second sum of (24), we again use Equation (23). Since $x_u(p'_r) > x_u(p_r)$ it follows that $\tilde{\delta}(p_r, p'_r, u, \hat{U}^1_{3-r}) = \sum_{v\in\hat{U}^1_{3-r}} w_{uv}(x_u(p'_r) - x_u(p_r)) = g^{\text{left}}_u L_s = \delta_4(u)$. Applying (23) to $\hat{U}^2_{3-r}$ gives

$$\begin{aligned}\tilde{\delta}(p_r, p'_r, u, \hat{U}^2_{3-r}) = \sum_{v\in\hat{U}^2_{3-r}} w_{uv}((x_u(p_r) + L_s - x_v(p_{3-r})) - (x_v(p_{3-r}) - x_u(p_r))) = \\ (L_s + 2x_u(p_r)) \sum_{v\in\hat{U}^2_{3-r}} w_{uv} - 2 \sum_{v\in\hat{U}^2_{3-r}} w_{uv} x_v(p_{3-r}).\end{aligned} \tag{25}$$

Since $L_s = \lambda^r_{\rho(s)}$, the first sum in the rightmost hand side of (25) is equal to $A^{\text{right}}_{u,\rho(s)}$ and the second sum is equal to $B^{\text{right}}_{u,\rho(s)}$. Thus $\tilde{\delta}(p_r, p'_r, u, \hat{U}^2_{3-r}) = \delta_5(u)$. Finally, for $\hat{U}^3_{3-r}$ we have $\tilde{\delta}(p_r, p'_r, u, \hat{U}^3_{3-r}) = \sum_{v\in\hat{U}^3_{3-r}} w_{uv}(x_u(p_r) - x_u(p'_r)) = -L_s(\sum_{v\in U_{3-r}\setminus\hat{U}^1_{3-r}} w_{uv} - \sum_{v\in\hat{U}^2_{3-r}} w_{uv}) = -L_s(g^{\text{right}}_u - A^{\text{right}}_{u,\rho(s)}) = -\delta_6(u)$. Substituting expressions for $\tilde{\delta}(p_r, p'_r, s, U^i_{3-r})$, $i = 1, 2, 3$, and $\tilde{\delta}(p_r, p'_r, u, \hat{U}^i_{3-r})$, $i = 1, 2, 3$, into (24) we arrive at Equation (9).

Suppose now that facility $s \in U_r$ is interchanged with its right neighbor u (see Fig. 4). In this case, we partition U_{3-r} into $U^1_{3-r} = \{p_{3-r}(i) \mid i = 1, \dots, z_s\}$, $U^2_{3-r} = H'_{3-r,su}$, and $U^3_{3-r} = U_{3-r} \setminus (U^1_{3-r} \cup U^2_{3-r})$. In Fig. 4, $U^1_{3-r} = \{p_2(1), p_2(2)\}$, $U^2_{3-r} = \{p_2(3), p_2(4), p_2(5)\}$, and $U^3_{3-r} = \{p_2(6), p_2(7), p_2(8)\}$. Arguing as in previous case we find that $\tilde{\delta}(p_r, p'_r, s, U^1_{3-r}) = \sum_{v\in U^1_{3-r}} w_{sv}(x_s(p'_r) - x_s(p_r)) = g^{\text{left}}_s L_u = \delta'_1(s)$, $\tilde{\delta}(p_r, p'_r, s, U^2_{3-r}) = \sum_{v\in U^2_{3-r}} w_{sv}((x_s(p_r) + L_u - x_v(p_{3-r})) - (x_v(p_{3-r}) - x_s(p_r))) = \delta'_2(s)$, and $\tilde{\delta}(p_r, p'_r, s, U^3_{3-r}) = \sum_{v\in U^3_{3-r}} w_{sv}(x_s(p_r) - x_s(p'_r)) = -L_u(\sum_{v\in U_{3-r}\setminus U^1_{3-r}} w_{sv} - \sum_{v\in U^2_{3-r}} w_{sv}) = -L_u(g^{\text{right}}_s - \sum_{v\in H'_{3-r,su}} w_{sv}) = -\delta'_3(s)$. To compute the change in the objective function value incurred by moving facility u, we partition U_{3-r} into $\hat{U}^1_{3-r} = \{p_{3-r}(i) \mid i = z_u + 1, \dots, n_{3-r}\}$, $\hat{U}^2_{3-r} = \{v \in U_{3-r} \mid x_u(p_r) - L_s < x_v(p_{3-r}) \leqslant x_u(p_r)\}$, and $\hat{U}^3_{3-r} = U_{3-r} \setminus (\hat{U}^1_{3-r} \cup \hat{U}^2_{3-r})$. In Fig. 4, $\hat{U}^1_{3-r} = \{p_2(7), p_2(8)\}$, $\hat{U}^2_{3-r} = \{p_2(2), \dots, p_2(6)\}$, and $\hat{U}^3_{3-r} = \{p_2(1)\}$. For these sets, we have $\tilde{\delta}(p_r, p'_r, u, \hat{U}^1_{3-r}) = \sum_{v\in\hat{U}^1_{3-r}} w_{uv}(x_u(p_r) - x_u(p'_r)) = g^{\text{right}}_u L_s = \delta'_4(u)$, $\tilde{\delta}(p_r, p'_r, u, \hat{U}^2_{3-r}) = \sum_{v\in\hat{U}^2_{3-r}} w_{uv}((x_v(p_{3-r}) - x_u(p_r) + L_s) - (x_u(p_r) - x_v(p_{3-r}))) = (L_s - 2x_u(p_r)) \sum_{v\in\hat{U}^2_{3-r}} w_{uv} + 2\sum_{v\in\hat{U}^2_{3-r}} w_{uv} x_v(p_{3-r}) = (L_s - 2x_u(p_r))A^{\text{left}}_{u,\rho(s)} + 2B^{\text{left}}_{u,\rho(s)} = \delta'_5(u)$, and $\tilde{\delta}(p_r, p'_r, u, \hat{U}^3_{3-r}) = \sum_{v\in\hat{U}^3_{3-r}} w_{uv}(x_u(p'_r) - x_u(p_r)) = -L_s(\sum_{v\in U_{3-r}\setminus\hat{U}^1_{3-r}} w_{uv} - \sum_{v\in\hat{U}^2_{3-r}} w_{uv}) = -L_s(g^{\text{left}}_u - A^{\text{left}}_{u,\rho(s)}) = -\delta'_6(u)$. To finish the proof, observe that putting $\tilde{\delta}(p_r, p'_r, s, U^i_{3-r})$, $i = 1, 2, 3$, and $\tilde{\delta}(p_r, p'_r, u, \hat{U}^i_{3-r})$, $i = 1, 2, 3$, together yields (16). □

Now, we return to Equations (7) and (8). We consider the case of $i = 1$. In order to apply (7) and (8), we need to compute $\delta^1(p_r, s, u)$. For $r \in \{1, 2\}$ and $k \in \{1, \dots, n_r - 1\}$, we define $e^r_k = \sum_{i=1}^{k} \sum_{j=k+1}^{n_r} w_{p_r(i)p_r(j)}$. These sums can be obtained recursively as follows:

$$e^r_k = e^r_{k-1} + \hat{h}^r_v - h^r_v, k = 1, \dots, n_r - 1, \tag{26}$$

where $v = p_r(k)$, $h^r_v = \sum_{i=1}^{k-1} w_{vp_r(i)}$, $\hat{h}^r_v = \sum_{i=k+1}^{n_r} w_{vp_r(i)}$, and, by convention, $e^r_0 = 0$. Assuming, in addition, that $e^r_{n_r} = 0$, we can compute the differences

$$\bar{e}^r_k = e^r_k - e^r_{k-1}, k = 1, \dots, n_r. \tag{27}$$

To efficiently calculate $\delta^1(p_r, s, u)$, we use the same formulas that were provided for the SRFLP in [29]. The following statement is an adaptation of Proposition 6 in [29] for the PROP.

Proposition 2. *Given a permutation $p_r \in \Pi^r(n_r)$, let $s = p_r(k)$ be the facility in position k, where $1 \leqslant k \leqslant n_r$. Then, for $u = p_r(l)$ with l changing from $k-1$ to 1, we have*

$$\delta^1(p_r, s, u) = \delta^1(p_r, s, p_r(l+1)) + L_s(w_{su} - \vec{e}^r_l) + L_u(\beta_l + \beta_{l+1} + \vec{e}^r_k), \tag{28}$$

where $\beta_l = \beta_{l+1} + w_{su}$ and initially $\delta^1(p_r, s, s)$ and β_k are equal to zero.

For $u = p_r(l)$ with l changing from $k+1$ to n_r and initial conditions as above, we have

$$\delta^1(p_r, s, u) = \delta^1(p_r, s, p_r(l-1)) + L_s(w_{su} + \vec{e}^r_l) + L_u(\beta_{l-1} + \beta_l - \vec{e}^r_k), \tag{29}$$

where $\beta_l = \beta_{l-1} + w_{su}$.

Armed with Propositions 1 and 2, we can now present our local search algorithm for the PROP. For the sake of better exposition, we split the algorithm into several parts. The pseudocode of the top level of LS appears as Algorithm 1. There, we use a few additional notations. Let $u = p_r(k)$, $r \in \{1,2\}$, $k \in \{1,\dots,n_r\}$, and $q \in \{1,\dots,m_r\}$. Provided $k > 1$, we denote by Φ^{left}_{uq} the cardinality of the set $\{i \mid i = 1,\dots,k-1, \rho(p_r(i)) = q\}$. Similarly, provided $k < n_r$, we define $\Phi^{\text{right}}_{uq} = |\{i \mid i = k+1,\dots,n_r, \rho(p_r(i)) = q\}|$. Under the same restrictions on k, we also define $\varphi^{\text{left}}_u = \max\{q \mid \Phi^{\text{left}}_{uq} > 0\}$ and $\varphi^{\text{right}}_u = \max\{q \mid \Phi^{\text{right}}_{uq} > 0\}$. We use Φ^{left}_{uq}, Φ^{right}_{uq}, φ^{left}_u, and φ^{right}_u to efficiently initialize and update A^{left}_{uq}, B^{left}_{uq}, A^{right}_{uq}, and B^{right}_{uq}.

Algorithm 1 Local search.

```
    local_search(p1, p2)
Input: Solution (p1, p2), where p1 ∈ Π1(n1) and p2 ∈ Π2(n2) are permutations.
Output: Locally optimal solution (p1, p2) and its objective function value.
 1: for r = 1, 2 do
 2:    Using (26)–(27), compute e_k^r, k = 1, ..., n_r
 3:    Compute x_u(p_r) for each u ∈ U_r
 4: end for
 5: for r = 1, 2 do
 6:    Compute z_u for each u ∈ U_r
 7:    for k = 1, ..., n_r and u = p_r(k) do
 8:       Compute g_u^left and g_u^right
 9:       Compute Φ_ui^left, i = 1, ..., m_r, and φ_u^left
10:       if k > 1 then left_side(r, u) end if
11:       Compute Φ_ui^right, i = 1, ..., m_r, and φ_u^right
12:       if k < n_r then right_side(r, u) end if
13:    end for
14: end for
15: Compute f = F(p1, p2)
16: Δmin ← −1;
17: while Δmin < 0 do
18:    (p1, p2, Δmin) ← explore_neighborhood(p1, p2);
19:    f ← f + Δmin;
20: end while
21: return solution (p1, p2) of value f
```

From the pseudocode, we see that our LS algorithm consists of two parts: initialization (Lines 1–15) and repetitive exploration of the insertion neighborhood of the current solution (p_1, p_2) (Lines 16–20). The first loop (Lines 1–4) computes the values stored in the array $\{\vec{e}^r_1, \dots, \vec{e}^r_{n_r}\}$ and the x coordinate of each facility for the permutation p_r, $r = 1, 2$. Having the x coordinates of all facilities, we can easily calculate z_u for each facility $u \in U_r$, $r \in \{1,2\}$ (Line 6). First, we merge the permutations p_1 and p_2 into a single list of facilities sorted according to their x coordinates. Then, passing through this list, the value of z_u is initialized for each $u \in U_r$, $r \in \{1,2\}$. Clearly, this step runs in $O(n)$ time. Once z_u, $u \in U_r$, are ready, the sums g^{left}_u and g^{right}_u can be computed in a straightforward manner (Line 8). In the next step, Φ^{left}_{ui}, $i = 1,\dots,m_r$, and φ^{left}_u for $u = p_r(k)$ are calculated using $\rho(v_i)$ values for $v_i = p_r(i)$, $i = 1,\dots,k-1$ (Line 9). If $k > 1$, then the `left_side` procedure is triggered (Line 10). It computes A^{left}_{uq} and B^{left}_{uq} for each q with positive Φ^{left}_{uq}. The statements in Lines 11 and 12 are similar to those in Lines 9 and 10. In this case, facilities to the right of u are considered. The initialization stage ends with the computation of the objective value on the input solution (Line 15).

After initialization, the algorithm proceeds to search for a better solution than the initial one (Lines 16–20). In this process, a neighborhood exploration procedure is invoked at least once. The process stops when the current solution is found to be locally optimal. In this case, $\Delta_{\min}$ (returned in Line 18) is equal to zero. The output of the algorithm is the solution (p_1, p_2) and its objective function value f.

Algorithms 2 and 3 show the pseudocode of procedures `left_side` and `right_side`, respectively. The reason behind presenting a detailed description of `left_side` and `right_side` is to make the evaluation of the complexity of these procedures more straightforward. Suppose $q \geqslant 1$ and $q' \in \{q+1,\dots,m_r\}$ are such that $\Phi^{\text{left}}_{uq} > 0$ and $\Phi^{\text{left}}_{uq'} > 0$ for a facility $u \in U_r$. Then, from the definition of A^{left}_{uq} it follows that $A^{\text{left}}_{uq'} \geqslant A^{\text{left}}_{uq}$. Even more, if $A^{\text{left}}_{uq'} > A^{\text{left}}_{uq}$, then $A^{\text{left}}_{uq'}$ is obtained by adding one or more terms to A^{left}_{uq}.

Algorithm 2 Computation of A_{uq}^{left} and B_{uq}^{left}.

```
    left_side(r, u)
Input: Row r, facility u.
Output: A^left_uq, B^left_uq, q ∈ {1, ..., m_r | Φ^left_uq > 0}.
 1: a, b ← 0;
 2: j, j' ← z_u;
 3: for q = 1, ..., m_r such that Φ^left_uq > 0 do
 4:    x' ← x_u(p_r) − λ^r_q;
 5:    for i = j, j − 1, ..., 1 do
 6:       v ← p_{3−r}(i);
 7:       if x_v(p_{3−r}) > x' then
 8:          a ← a + w_uv;
 9:          b ← b + w_uv x_v(p_{3−r});
10:          j' ← i − 1;
11:       else
12:          j' ← i;
13:          Break from the loop
14:       end if
15:    end for
16:    A^left_uq ← a;
17:    B^left_uq ← b;
18:    j ← j';
19: end for
```

Algorithm 3 Computation of A_{uq}^{right} and B_{uq}^{right}.

```
    right_side(r, u)
Input: Row r, facility u.
Output: A^right_uq, B^right_uq, q ∈ {1, ..., m_r | Φ^right_uq > 0}.
 1: a, b ← 0;
 2: j, j' ← z_u + 1;
 3: for q = 1, ..., m_r such that Φ^right_uq > 0 do
 4:    x' ← x_u(p_r) + λ^r_q;
 5:    for i = j, j + 1, ..., n_{3−r} do
 6:       v ← p_{3−r}(i);
 7:       if x_v(p_{3−r}) < x' then
 8:          a ← a + w_uv;
 9:          b ← b + w_uv x_v(p_{3−r});
10:          j' ← i + 1;
11:       else
12:          j' ← i;
13:          Break from the loop
14:       end if
15:    end for
16:    A^right_uq ← a;
17:    B^right_uq ← b;
18:    j ← j';
19: end for
```

The same property holds for B_{uq}^{left}, A_{uq}^{right}, and B_{uq}^{right}. Therefore, it is natural to apply an incremental algorithm to calculate these quantities. We notice that there is no need to compute A_{uq}^{left} and B_{uq}^{left} for $q \in \{1, \dots, m_r\}$ such that $\Phi_{uq}^{\text{left}} = 0$. Indeed, A_{uq}^{left} and B_{uq}^{left} are required only when a facility v with $L_v = \lambda_q^r$ is moved to a different position and during this movement is interchanged with facility u being the right neighbor of v (as depicted in Fig. 4 for $v = s$ and seen in Equations (21) and (22)). This, however, never happens because all the facilities of length λ_q^r appear at the right side of facility u in the permutation p_r. Such a situation is indicated with $\Phi_{uq}^{\text{left}} = 0$.

Algorithm 2 starts by initializing variables a and b, which will ultimately determine A_{uq}^{left} and B_{uq}^{left} for each $q \in \{1, \dots, m_r\}$ with positive Φ_{uq}^{left} (Lines 16 and 17). For every such q, the algorithm considers the semiopen interval $X_q = (x', x_u(p_r)]$ whose right end is fixed and left end depends on q (and is set in Line 4). The goal is to identify all facilities in U_{3-r} whose x coordinates fall in the interval X_q (Lines 6 and 7). For every such facility, denoted v, the sums a and b (and indirectly A_{uq}^{left} and B_{uq}^{left}) are increased by adding w_{uv} and $w_{uv}x_v(p_{3-r})$, respectively (Lines 8 and 9). By executing the inner "for" loop repeatedly, the sums A_{uq}^{left} and B_{uq}^{left} are eventually obtained. Execution of this loop for the next value of q with $\Phi_{uq}^{\text{left}} > 0$ starts from the index stored in the variable j' (Lines 10, 12, and 18). The first execution of the inner loop starts from $j = z_u$ (Line 2). This choice is justified by the fact that the x coordinates of the facilities $p_{3-r}(j)$ with j greater than z_u lie outside the interval X_q for all q. Algorithm 3 shows the pseudocode of `right_side`. This procedure bears a great similarity to the procedure `left_side` just discussed. The main difference is that facilities in `right_side`

are processed in the direction from left to right. From the pseudocode, it can be seen that both `left_side` and `right_side` have $O(n)$ time complexity.

As mentioned before, the main component of our LS algorithm is a neighborhood exploration procedure. The pseudocode of this procedure is presented in Algorithm 4. All $n_r - 1$ possible insertion positions for facility $s \in U_r$ are enumerated systematically. First, facility s is moved to the left (Line 5) and then to the right (Line 6). At each step, the change in objective value is calculated. The variable $\Delta_{\min}$ stands for the maximum cost reduction achieved during the search process. A negative value of $\Delta_{\min}$ means detecting an insertion move that leads to a better solution. Such a move is stored by saving the row r' containing the selected facility, the current position k' of this facility in the permutation $p_{r'}$, and its new position l'. If $\Delta_{\min} < 0$, then the input solution is replaced by the best solution in its neighborhood (Line 11). Otherwise, the input solution is found to be locally optimal, and $\Delta_{\min} = 0$ is returned. In the former case, the procedure `update_info` is triggered (Line 12). It updates arrays and matrices needed for scanning of the neighborhood of the new solution at the next invocation of `explore_neighborhood` (Line 18 in Algorithm 1).

Algorithm 4 Exploration of the insertion neighborhood.

`explore_neighborhood`(p_1, p_2)
Input: Pair of permutations (p_1, p_2).
Output: Possibly improved solution (p_1, p_2) and the change in the objective function value $\Delta_{\min}$.
1: $\Delta_{\min} \leftarrow 0$;
2: **for** $r = 1, 2$ **do**
3: **for** $k = 1, \ldots, n_r$ **do**
4: $s \leftarrow p_r(k)$;
5: $(\Delta_{\min}, r', k', l') \leftarrow$ `move_left`$(r, s, k, \Delta_{\min})$;
6: $(\Delta_{\min}, r', k', l') \leftarrow$ `move_right`$(r, s, k, \Delta_{\min})$;
7: **end for**
8: **end for**
9: **if** $\Delta_{\min} < 0$ **then**
10: $s \leftarrow p_{r'}(k')$;
11: Update $p_{r'}$ by inserting facility s in position l'
12: `update_info`$(p_1, p_2, r', s, k', l')$
13: **end if**
14: **return** p_1, p_2, and $\Delta_{\min}$

Algorithm 5 Moving the facility to the left.

`move_left`$(r, s, k, \Delta_{\min})$
Input: Row r, facility s, its location k, and the best move gain $\Delta_{\min}$.
Output: Best move gain $\Delta_{\min}$ and, possibly, r', k', and l'.
1: $\Delta \leftarrow 0$;
2: $x' \leftarrow x_s(p_r)$;
3: $z \leftarrow z_s$;
4: $\tilde{g}^{\text{left}} \leftarrow g_s^{\text{left}}$;
5: $\tilde{g}^{\text{right}} \leftarrow g_s^{\text{right}}$;
6: $\beta \leftarrow 0$;
7: **for** l changing from $k-1$ down to 1 and $u = p_r(l)$ **do**
8: $\delta \leftarrow \tilde{g}^{\text{right}} L_u$; // implements (10)
9: **while** $z > 0$ and $x_v(p_{3-r}) \geqslant x' - L_u$ for $v = p_{3-r}(z)$ **do**
10: $\delta \leftarrow \delta + (L_u - 2(x' - x_v(p_{3-r})))w_{sv}$; // implements (11)
11: Add w_{sv} to $\tilde{g}^{\text{right}}$ and subtract from $\tilde{g}^{\text{left}}$
12: Decrement z by 1
13: **end while**
14: **if** $z > 0$ **then** $\delta \leftarrow \delta - \tilde{g}^{\text{left}} L_u$; **end if** // implements (12)
15: Increase δ by $\delta_4(u) + \delta_5(u) - \delta_6(u)$ calculated by (13)–(15)
16: $\beta' \leftarrow \beta$; // β' represents β_{l+1} in (28)
17: $\beta \leftarrow \beta + w_{su}$; // β represents β_l in (28)
18: Increase δ by $L_s(w_{su} - \bar{e}_l^r) + L_u(\beta + \beta' + \bar{e}_k^r)$ // Proposition 2
19: $\Delta \leftarrow \Delta + \delta$; // implements (6)
20: **if** $\Delta < \Delta_{\min}$ **then**
21: Assign Δ to $\Delta_{\min}$, r to r', k to k', and l to l'
22: **end if**
23: $x' \leftarrow x' - L_u$;
24: **end for**
25: **return** $\Delta_{\min}$ and, possibly, r', k', and l'

The neighbors of the solution (p_1, p_2) are evaluated by executing `move_left` and `move_right` in Lines 5 and 6 of `explore_neighborhood`. The pseudocode of `move_left` is given in Algorithm 5. There are two important points to note about this procedure. First, its implementation is based on Propositions 1 and 2. Second, facility s is moving from the kth position to the first position in the permutation p_r. As a result, the x coordinate of the facility s and other information about s are changing. Because

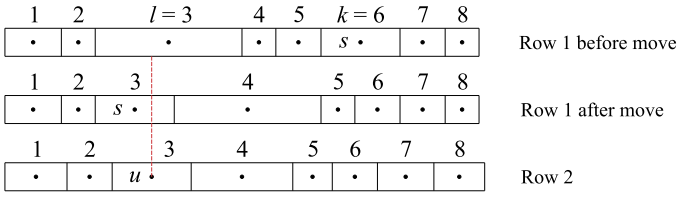

Fig. 5. Inserting facility s to position $l = 3$.

of that, some special variables for s are used in the algorithm: x' stands for the current x coordinate of the facility s, and z, $\tilde{g}^{\text{left}}$, and $\tilde{g}^{\text{right}}$ represent the current values of z_s, g_s^{left}, and g_s^{right}, respectively. These variables are initialized in Lines 2–5 and updated in Lines 11, 12, and 23. The variable β represents β_l used in Proposition 2. The aim of `move_left` is to calculate the change in objective value, Δ, for each possible position of the facility s to the left of its initial position. To accomplish this task, the incremental gain change δ is calculated (Lines 8–18) and Equation (6) is applied (Line 19) a number of times. Using Propositions 1 and 2, the δ value is computed in several steps. Specifically, δ is initialized using Equation (10) (Line 8). The loop in Lines 9–13 implements Equation (11). The termination condition of the loop checks whether $v \in H_{3-r,su}$. After updating $\tilde{g}^{\text{left}}$ within the loop, Equation (12) becomes $\delta_3(s) = \tilde{g}^{\text{left}} L_u$ if $z > 0$ or $\delta_3(s) = 0$ if $z = 0$. In the former case, $\delta_3(s)$ is subtracted from δ (Line 14). In the next step, $\delta_i(u)$, $i = 4, 5, 6$, are calculated via Equations (13)–(15) and used to update δ according to Equation (9) (Line 15). It can be noticed that Lines 8–15 of `move_left` yield $\delta^2(p_r, s, u)$ as given by Equation (9). Line 18 adds $\delta^1(p_r, s, u)$, as given by (28), to δ. When a new best move is found it is memorized (Lines 20–22). If this happens at least once, then the triplet (r', k', l') is returned. The pseudocode of `move_right` is shown in Algorithm 6. This procedure is very similar to `move_left` discussed above.

Algorithm 6 Moving the facility to the right.

`move_right`$(r, s, k, \Delta_{\min})$
Input: Row r, facility s, its location k, and the best move gain $\Delta_{\min}$.
Output: Best move gain $\Delta_{\min}$ and, possibly, r', k', and l'.
1: $\Delta \leftarrow 0$;
2: $x' \leftarrow x_s(p_r)$;
3: $z \leftarrow z_s + 1$;
4: $\tilde{g}^{\text{left}} \leftarrow g_s^{\text{left}}$;
5: $\tilde{g}^{\text{right}} \leftarrow g_s^{\text{right}}$;
6: $\beta \leftarrow 0$;
7: **for** $l = k + 1, \ldots, n_r$ and $u = p_r(l)$ **do**
8: $\delta \leftarrow \tilde{g}^{\text{left}} L_u$; // implements (17)
9: **while** $z \leqslant n_{3-r}$ and $x_v(p_{3-r}) \leqslant x' + L_u$ for $v = p_{3-r}(z)$ **do**
10: $\delta \leftarrow \delta + (L_u - 2(x_v(p_{3-r}) - x'))w_{sv}$; // implements (18)
11: Add w_{sv} to $\tilde{g}^{\text{left}}$ and subtract from $\tilde{g}^{\text{right}}$
12: Increment z by 1
13: **end while**
14: **if** $z \leqslant n_{3-r}$ **then** $\delta \leftarrow \delta - \tilde{g}^{\text{right}} L_u$; **end if** // implements (19)
15: Increase δ by $\delta_4'(u) + \delta_5'(u) - \delta_6'(u)$ calculated by (20)–(22)
16: $\beta' \leftarrow \beta$; // β' represents β_{l-1} in (29)
17: $\beta \leftarrow \beta + w_{su}$; // β represents β_l in (29)
18: Increase δ by $L_s(w_{su} + \bar{e}_l^r) + L_u(\beta + \beta' - \bar{e}_k^r)$ // Proposition 2
19: $\Delta \leftarrow \Delta + \delta$; // implements (6)
20: **if** $\Delta < \Delta_{\min}$ **then**
21: Assign Δ to $\Delta_{\min}$, r to r', k to k', and l to l'
22: **end if**
23: $x' \leftarrow x' + L_u$;
24: **end for**
25: **return** $\Delta_{\min}$ and, possibly, r', k', and l'

Having r', k', and l', the following x coordinates can be calculated:

$$x_{\min} = \begin{cases} \min(x_u(\tilde{p}_{r'}), x_s(p_{r'})) & \text{if } l' < k' \\ \min(x_s(\tilde{p}_{r'}), x_v(p_{r'})) & \text{if } l' > k', \end{cases} \tag{30}$$

$$x_{\max} = \begin{cases} \max(x_s(\tilde{p}_{r'}), x_v(p_{r'})) & \text{if } l' < k' \\ \max(x_u(\tilde{p}_{r'}), x_s(p_{r'})) & \text{if } l' > k', \end{cases} \tag{31}$$

where $u = \tilde{p}_{r'}(l')$, $v = p_{r'}(k')$, and $\tilde{p}_{r'}$ (respectively, $p_{r'}$) is the permutation for row r' that existed before (respectively, after) the insertion operation in Line 11 of Algorithm 4 was performed. Basically, $x_{\min}$ is the smallest x coordinate (either in $\tilde{p}_{r'}$ or $p_{r'}$) among all facilities that were affected by this operation. The x coordinate $x_{\max}$ is defined analogously. Both $x_{\min}$ and $x_{\max}$ are used in the procedure `update_info` (applied in Line 12 of Algorithm 4). Its pseudocode is given in Algorithm 7.

Algorithm 7 Updating arrays and matrices.

`update_info`(p_1, p_2, r, s, k, l)
Input: Solution (p_1, p_2), row r, facility s, its old position k, and new position l.
Output: Updated arrays and matrices used in the local search.
1: $\kappa \leftarrow \min(k, l)$;
2: $\psi \leftarrow \max(k, l)$;
3: $z'_u \leftarrow z_u$ for $u \in U_{3-r} \cup \{p_r(i) \mid i = \kappa, \dots, \psi\}$
4: Compute z_u for $u \in \{p_r(i) \mid i = \kappa, \dots, \psi\}$
5: Compute z_u for each $u \in U_{3-r}$ such that $\max(\kappa - 1, 1) \leqslant z'_u \leqslant \psi$
6: **for** $i = \kappa, \dots, \psi$ and $u = p_r(i)$ such that $z_u \neq z'_u$ **do**
7: Update g_u^{left} and g_u^{right}
8: **end for**
9: **for** $i = 1, \dots, n_{3-r}$ and $u = p_{3-r}(i)$ such that $\kappa - 1 \leqslant z'_u \leqslant \psi$ **do**
10: Update g_u^{left} and g_u^{right}
11: **end for**
12: **for** each $i = \kappa, \dots, \psi$ except $i = l$ **do**
13: $u \leftarrow p_r(i)$;
14: Update $\Phi^{\text{left}}_{u,\rho(s)}$, $\Phi^{\text{right}}_{u,\rho(s)}$ and possibly φ_u^{left} and φ_u^{right}
15: Update $\Phi^{\text{left}}_{s,\rho(u)}$, $\Phi^{\text{right}}_{s,\rho(u)}$ and possibly φ_s^{left} and φ_s^{right}
16: **if** $i > 1$ **then** `left_side`(r, u) **end if**
17: **if** $i < n_r$ **then** `right_side`(r, u) **end if**
18: **end for**
19: **if** $l > 1$ **then** `left_side`(r, s) **end if**
20: **if** $l < n_r$ **then** `right_side`(r, s) **end if**
21: Calculate $x_{\min}$ by (30) and $x_{\max}$ by (31)
22: **for** $i = 2, \dots, n_{3-r}$ and $u = p_{3-r}(i)$ **do**
23: $j \leftarrow \varphi_u^{\text{left}}$;
24: **if** $x_u(p_{3-r}) \geqslant x_{\min}$ and $x_u(p_{3-r}) - \lambda_j^{3-r} < x_{\max}$ **then**
25: `left_side`$(3 - r, u)$
26: **end if**
27: **end for**
28: **for** $i = n_{3-r} - 1$ to 1 by -1, and $u = p_{3-r}(i)$ **do**
29: $j \leftarrow \varphi_u^{\text{right}}$;
30: **if** $x_u(p_{3-r}) < x_{\max}$ and $x_u(p_{3-r}) + \lambda_j^{3-r} > x_{\min}$ **then**
31: `right_side`$(3 - r, u)$
32: **end if**
33: **end for**
34: Using (26) and (27), update $\bar{e}_i^r$, $i = \kappa, \dots, \psi$

The procedure `update_info` starts by setting values for κ and ψ that define an interval in the permutation p_r. Each facility $p_r(i)$ with $i \in [\kappa, \psi]$ was moved to a new location. For each such facility u and each facility in the opposite row, the current value of z_u is saved as z'_u (Line 3). These values are needed to compute z_u for each $u \in U_{3-r}$ (Line 5) and update g_u^{left} and g_u^{right} for each $u \in U_{3-r} \cup \{p_r(i) \mid i \in [\kappa, \psi]\}$ (Lines 6–11). In Line 4, z_u is computed for $u \in \{p_r(i) \mid i \in [\kappa, \psi]\}$. This step of the algorithm follows the same strategy as used in the initialization phase of LS (Line 6 of Algorithm 1). Notice that z_u does not change for $u = p_r(i)$ and i lying outside the interval $[\kappa, \psi]$. The same is true for $u \in U_{3-r}$ such that either $z'_u > \psi$ or $z'_u < \kappa - 1$ (provided $\kappa > 2$). This does not necessarily hold for $u \in U_{3-r}$ with $z'_u = \kappa - 1$. A layout example, illustrating this case, is shown in Fig. 5. In this example, $r = 1$, $k = 6$, $\kappa = l = 3$, $u \in U_2$, $z'_u = \kappa - 1 = 2$, and $z_u = 3 > z'_u$. Again, z_u for $u \in U_{3-r}$ can be computed (Line 5) using the same approach as in the initialization phase of LS. Once z_u and z'_u are ready, it is possible to update g_u^{left} and g_u^{right} in a straightforward way. This is done for each facility $u = p_r(i)$ such that $i \in [\kappa, \psi]$ and, trivially, $z_u \neq z'_u$ (Lines 6–8) as well as for each facility $u \in U_{3-r}$ with $z'_u \in [\kappa - 1, \psi]$ (Lines 9–11). The next loop (Lines 12–18) starts by updating $\Phi^{\text{left}}_{u,\rho(s)}$, $\Phi^{\text{right}}_{u,\rho(s)}$, and possibly φ_u^{left} and φ_u^{right}. If $l < k$, for example, then $\Phi^{\text{left}}_{u,\rho(s)}$ for $u = p_r(i)$, $i = l + 1, \dots, k$, is increased by 1 and $\Phi^{\text{right}}_{u,\rho(s)}$ is decreased by 1. The value of φ_u^{right} is updated if it equals $\rho(s)$ and $\Phi^{\text{right}}_{u,\rho(s)}$ becomes equal to zero, and φ_u^{left} is updated if $\rho(s) > \varphi_u^{\text{left}}$. The same operations are performed with respect to facility s (Line 15). At the end of the loop, `left_side` (if $i > 1$) and `right_side` (if $i < n_r$) procedures are executed for facility u (Lines 16 and 17). They return updated A_{uq}^{left}, B_{uq}^{left}, A_{uq}^{right}, and B_{uq}^{right} for $q = 1, \dots, m_r$. The same procedures are applied to s only when operations in Line 15 for all $u = p_r(i)$ with $i \in \{\kappa, \dots, \psi\} \setminus \{l\}$ have been completed (Lines 19 and 20). The values of A_{uq}^{left} and B_{uq}^{left} for $u \in U_{3-r}$ are computed by invoking `left_side` within the loop 22–27. There is no need to recompute them for u satisfying one of the following inequalities: $x_u(p_{3-r}) < x_{\min}$ or $x_u(p_{3-r}) - \lambda_j^{3-r} \geqslant x_{\max}$, where $j = \varphi_u^{\text{left}}$ and $x_{\min}$ and $x_{\max}$ are defined by (30) and (31), respectively. The modified values of A_{uq}^{right} and B_{uq}^{right} for $u \in U_{3-r}$ are computed inside the loop 28–33. In the last step of the algorithm, $\bar{e}_i^r$ for $i = \kappa, \dots, \psi$ are updated using Equations (26) and (27) (Line 34). It can be noticed that $\bar{e}_i^r$ do not change outside the interval $[\kappa, \psi]$.

In closing this section, we make an observation about the time complexity of exploring the insertion neighborhood N. In the statement below, we assume that the exploration of the whole neighborhood is understood as an enumeration of the objective function values of all solutions in the neighborhood $N(p)$ of a solution p.

Theorem 1. *Given a solution p of the PROP, the neighborhood $N(p)$ can be explored in optimal time $\Theta(n^2)$.*

Proof. The lower bound on the number of operations required to explore the insertion neighborhood is $\Omega(n^2)$. It follows from the fact that the size of this neighborhood is $n_1(n_1-1)+n_2(n_2-1)$. A matching upper bound is provided by the procedure `explore_neighborhood` presented in this section. To show this, we first consider the procedure `move_left`. Its inner loop (Lines 9–13 in Algorithm 5) is executed at most $z_s < n$ times. Since this loop and all statements in Lines 8 and 14–23 of the outer loop perform $O(1)$ operations, it follows that the time complexity of `move_left` (and certainly of `move_right`) is $O(n)$. This implies that the double-nested loop of `explore_neighborhood` (Lines 2–8) is executed in $O(n^2)$ time. It remains to evaluate the performance of the `update_info` procedure (triggered in Line 12 of Algorithm 4). The new values of z_u in Lines 4 and 5 of this procedure are computed in the same way as in the initialization stage of LS. As remarked before, this can be done in $O(n)$ time. The values of g_u^{left} and g_u^{right} are updated (Lines 6–11) by performing $O(n^2)$ operations. Updating the Φ and φ values in Lines 14 and 15 takes $O(1)$ and $O(n)$ time at worst, respectively. It was observed before that the computational complexity of both `left_side` and `right_side` is $O(n)$. Putting the above two points together, we see that the loop spanning Lines 12–18 runs in $O(n^2)$ time. The same upper bound also holds for the next two loops (Lines 22–33). Finally, in Line 34, the arrays $\bar{e}^r$, $r=1,2$, can be updated in time $O(n)$. We thus have shown that the time complexity of `update_info`, and hence of `explore_neighborhood`, is $O(n^2)$. □

3. A memetic algorithm

The local search technique of the previous section can be incorporated into a variety of metaheuristic algorithms for solving the PROP. An important class of such algorithms are categorized as evolutionary methods. They have been applied with great success to numerous combinatorial optimization problems in diverse fields. In this section, as a case study, we present a memetic algorithm for the PROP. We have chosen MA because this algorithm already exists for a long time and has shown excellent performance in solving many combinatorial optimization problems. The MA is a population-based stochastic optimization technique. The population evolves over a series of generations to explore the search space. At each generation, the individuals in the population are combined to obtain one or more offspring. The latter are improved by means of a local search procedure. A new population is formed by choosing the best individuals from the pool of parents and offspring.

Algorithm 8 Memetic algorithm for the PROP.

```
    MA
 1: for r = 1,2 do
 2:    Construct sorted array {λ_i^r | i = 1,...,m_r}
 3:    Compute ρ(u) and g_u^tot for each u ∈ U_r
 4: end for
 5: Initialize population P
 6: Let (p_1^*, p_2^*) denote the best solution in P and f^* be its value
 7: while stop condition is not met do
 8:    (P', p_1^*, p_2^*, f^*) ← create_generation(P, p_1^*, p_2^*, f^*);
 9:    P ← update_population(P');
10: end while
11: Stop with (p_1^*, p_2^*) and f^*
```

Algorithm 8 shows the overall framework of our MA implementation for the PROP. The algorithm consists of two phases: initialization (Lines 1–6) and solution improvement (Lines 7–10). In the first phase, an array λ^r for each $r \in \{1,2\}$ is created. It consists of lengths of facilities in U_r sorted in ascending order. Inside the same loop, the mapping ρ between the set U_r and array λ^r is constructed. Its definition was given in the previous section. Additionally, the sum $g_u^{\text{tot}} = \sum_{v \in U_{3-r}} w_{uv}$ for each $u \in U_r$ is calculated. The arrays λ^r, $r=1,2$, the mapping ρ, and the sums g_u^{tot}, $u \in U$, are used by our LS algorithm. All this information is derived from the problem instance and, of course, does not change throughout the search process. In Line 5 of MA, the population P is initialized by repeatedly executing two steps. First, a random solution to the PROP is generated. Then, using this solution as a seed for the `local_search` procedure (given as Algorithm 1), a locally optimal solution is obtained. In the pseudocode of the algorithm, (p_1^*, p_2^*) and f^* denote the best solution in P and its objective function value, respectively. The loop of the second phase of MA (Lines 7–10) applies two procedures, `create_generation` and `update_population`, which are executed iteratively one after the other. The first of them generates a predefined number of offspring, which are added to the population. The resulting pool of individuals, denoted by P', is too large. Therefore, the `update_population` procedure is triggered. It forms a new population by selecting the best individuals from the pool. The loop is executed until a certain stop condition is met. In our experiments with MA, we used a termination criterion, where the evolution is stopped when the specified CPU time limit is exceeded.

The pseudocode of the procedure `create_generation` is given in Algorithm 9. In the pseudocode, the resulting set of offspring is denoted by P_{gen}. At the beginning, P_{gen} is empty (Line 1). The required number of offspring is controlled by the MA parameter M_{gen}. The offspring are generated inside the "while" loop spanning Lines 2 to 12 in Algorithm 9. First, two parents, p' and p'', are randomly chosen from the population and submitted as an input to a crossover operator. Different crossover operators have been proposed in literature on evolutionary algorithms. We have chosen the position-based crossover (PBX) proposed by Syswerda [84]. The empirical study in [85] showed that the PBX operator is effective in exploring the search space of permutation-based optimization

Algorithm 9 Offspring generation.

```
    create_generation(P, p1*, p2*, f*)
Input: Population P, generation size M_gen, best-so-far p_r*, r = 1,2, and f*.
Output: Pool of permutation pairs P' and possibly improved p_r*, r = 1,2, and f*.
 1: Initialize P_gen as empty set
 2: while |P_gen| < M_gen do
 3:    Randomly choose two individuals p' = (p'_1, p'_2), p'' = (p''_1, p''_2) ∈ P
 4:    With p' and p'' as parents, generate offspring (p_1, p_2)
 5:    (p_1, p_2, f) ← local_search(p_1, p_2);
 6:    if f < f* then
 7:       (p1*, p2*) ← (p_1, p_2); f* ← f;
 8:    end if
 9:    if (p_1, p_2) differs from each individual in P ∪ P_gen then
10:       Add (p_1, p_2) to P_gen
11:    end if
12: end while
13: return P' = P ∪ P_gen
```

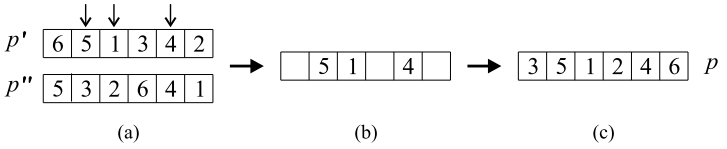

Fig. 6. Position-based crossover.

problems. The operation principle of the PBX is shown in Fig. 6. The parent individuals (permutations) p' and p'' are displayed in part (a) of the figure. In the first step, a set of positions in p' is randomly selected. In our example, this set contains the second, third, and the fifth positions. The genes in these positions are copied into the corresponding positions in the offspring (part (b) of Fig. 6). Next, the permutation p'' is scanned from left to right. The genes from p'' that are not present in the offspring are copied to it. The resulting offspring is shown in part (c) of the figure. In our algorithm for the PROP, the PBX operator is applied to each row of facilities independently. The size of the random set of positions in p_r, $r \in \{1,2\}$, is taken to be $\lceil n_r/2 \rceil$. After reproduction, the offspring is submitted to an improvement heuristic (Line 5), which is the `local_search` procedure presented in Algorithm 1. The returned solution is compared with the best-found solution. If the former is better than the latter, then it is memorized as a new best solution (Lines 6–8). Additionally, it is checked whether the new solution differs from each individual in parent and offspring populations. If so, then this solution is added to the set P_{gen} (Lines 9–11). The output of `create_generation` is the solution set P' that is obtained by merging parent (P) and offspring (P_{gen}) populations (Line 13).

Once offspring has been generated, the next step is to form a new population by selecting M_{pop} individuals from the combined pool of parent and offspring population (Line 9 of Algorithm 8). We implemented two versions of the procedure `update_population` used in this step. The selection mechanism in both versions is based on the fitness of individuals in the pool. However, one of the versions also takes into account the diversity of the surviving population. We define the similarity between solution $(p_1, p_2) \in P'$ and set $P \subset P'$ as follows:

$$D(p_1, p_2, P) = \min_{(p'_1, p'_2) \in P} S(p_1, p_2, p'_1, p'_2) = \min_{(p'_1, p'_2) \in P} \sum_{r=1}^{2} \sum_{i=1}^{n_r} |\pi_r(i) - \pi'_r(i)|, \tag{32}$$

where $\pi_r(i - n_1(r-1)) = k$ for k such that $p_r(k) = i$, and π'_r, $r \in \{1,2\}$, is defined in an analogous way for permutation p'_r. For illustration purposes, we calculate the similarity $S(p_1, p_2, p'_1, p'_2)$ between solution $p_1 = (6,5,1,3,4,2)$, $p_2 = (10,9,7,11,8)$ and solution $p'_1 = (5,6,3,2,4,1)$, $p'_2 = (10,11,9,7,8)$. Clearly, $\pi_1 = (3,6,4,5,2,1)$, $\pi_2 = (3,5,2,1,4)$, $\pi'_1 = (6,4,3,5,1,2)$, and $\pi'_2 = (4,5,3,1,2)$. Then, according to (32), we obtain $S(p_1, p_2, p'_1, p'_2) = (|3-6| + |6-4| + |4-3| + |5-5| + |2-1| + |1-2|) + (|3-4| + |5-5| + |2-3| + |1-1| + |4-2|) = 12$.

To define a criterion for choosing the best individual for inclusion into the next generation, we assume that the set P' is sorted in increasing order of objective function values. Let $R_{\text{obj}}(p_1, p_2)$ be the rank of solution $(p_1, p_2) \in P'$ in this sequence. We also assume that the set P consists of the already selected individuals. These individuals were removed from the pool P'. Suppose that P' is sorted in decreasing order of $D(p_1, p_2, P)$, and $R_{\text{dist}}(p_1, p_2)$ is the rank of solution $(p_1, p_2) \in P'$ in this sorted sequence. The fitness of solution $(p_1, p_2) \in P'$ is evaluated by the sum of $R_{\text{obj}}(p_1, p_2)$ and $R_{\text{dist}}(p_1, p_2)$. At each step, an individual with minimum value of this sum is moved from P' to P. The pseudocode of the first version of `update_population` procedure is given in Algorithm 10. The new population is initialized with the best solution in P' (Line 2). Other individuals are selected according to their ranks R_{obj} and R_{dist} (Line 7). Once selected, they are transferred from P' to P and removed from the sequences Q_{obj} and Q_{dist} (Lines 8 and 9). The process terminates when M_{pop} individuals are selected.

Another version of the procedure for generating a new population does not involve the computation of distances $D(p_1, p_2, P)$. It constructs the set P simply by selecting M_{pop} individuals from the pool P', which have the smallest objective function values (that is,

Algorithm 10 Generating a new population.

update_population(P')
Input: Pool of permutation pairs P', population size M_{pop}.
Output: Population P.
1: Sort P' in increasing order of objective function values
// This gives sequence Q_{obj}
2: Initialize P with $\tilde{p} = (\tilde{p}_1, \tilde{p}_2)$ such that $R_{obj}(\tilde{p}_1, \tilde{p}_2) = 1$
// $R_{obj}(p_1, p_2)$ is the rank of (p_1, p_2) in Q_{obj}
3: Remove $\tilde{p}$ from P' and Q_{obj}
4: Compute $D(p_1, p_2, P)$ for each $(p_1, p_2) \in P'$ // Equation (32)
5: Sort P' in decreasing order of $D(p_1, p_2, P)$ // This gives sequence Q_{dist}
6: **while** $|P| < M_{pop}$ **do**
7: $(\tilde{p}_1, \tilde{p}_2) \leftarrow \arg\min_{(p_1,p_2)\in P'}(R_{obj}(p_1, p_2) + R_{dist}(p_1, p_2))$;
// $R_{dist}(p_1, p_2)$ is the rank of (p_1, p_2) in Q_{dist}
8: Transfer $(\tilde{p}_1, \tilde{p}_2)$ from P' to P
9: Remove $(\tilde{p}_1, \tilde{p}_2)$ from Q_{obj} and Q_{dist}
10: Update $D(p_1, p_2, P)$, $(p_1, p_2) \in P'$, and the sequence Q_{dist}
11: **end while**
12: **return** P

P is composed of the first M_{pop} solutions in the sequence Q_{obj}). We refer to this version of the procedure as update_population$'$. The memetic algorithm with update_population$'$ used in Line 9 of Algorithm 8 will be referred to as MA$'$.

4. Computational experiments

This section is devoted to the performance assessment of the proposed memetic algorithm for the PROP. We evaluate the effectiveness of the algorithm by conducting two major experiments. In the first experiment, we compare MA against state-of-the-art algorithms proposed in recent literature. In the second experiment, we compare the two versions of the presented algorithm on a set of large-scale PROP instances.

4.1. Setup and datasets

The proposed memetic algorithm was implemented in C++ and ran on a PC equipped with 2.90 GHz Intel Core i5-9400F processor. The following datasets were used to test the MA performance:

(a) AV25 [17] containing 5 instances of size 25;
(b) P24 [8] containing 5 instances of size 24;
(c) sko56 [86] containing 5 instances of size 56;
(d) AKV_60 [16] containing 5 instances of size 60;
(e) AKV_70 [16] containing 5 instances of size 70;
(f) 26 instances with $250 \leqslant n \leqslant 500$ used in [33] for the SRFLP. This dataset as well as the source code of the presented algorithm are made publicly available.[1]

We note that the first n_1 facilities listed in a SRFLP instance are placed in the first row (assigned to the set U_1) and the remaining $n - n_1$ facilities are placed in the second row (assigned to the set U_2).

Our main experimental results are presented in Sections 4.3 and 4.4. In these experiments, we performed 20 independent runs of MA on each test instance. The two main measures used to assess the quality of the proposed algorithm are the best solution value found by MA and the mean solution value obtained by MA over all 20 runs.

4.2. Parameter settings

The two control parameters of the MA are the population size M_{pop} and generation size M_{gen}. To find out the best settings of these parameters, we performed a full factorial experiment by testing all combinations of the M_{pop} and M_{gen} values. The list of candidate values for parameter M_{pop} was $\{25, 50, 75, 100\}$ and that for parameter M_{gen} was $\{10, 20, 30, 40, 50\}$. These values were established after preliminary numerical experiments. For each combination (M_{pop}, M_{gen}), we ran MA on a training data set comprised of 15 problem instances. We used 5 instances with $n = 200$, 5 instances with $n = 250$, and 5 instances with $n = 300$. These PROP instances were constructed using the same generator that was used for the experimentation in [33]. The number of facilities in the first row, n_1, was fixed at $n/2$. The solution quality was measured as a relative difference between two objective values

$$c = (f - f^*)/f^*, \tag{33}$$

[1] https://drive.google.com/drive/folders/1ELhRhY_HFehpLovholFepBa-jwNoKpWc?usp=sharing.

Table 2
Average relative difference over a training dataset for different combinations of MA parameters.

M_{pop}	M_{gen}					Average
	10	20	30	40	50	
25	8700	4451	2453	3332	3944	4576
50	6110	4212	2794	1560	2551	3445
75	4017	3665	3223	**1391**	3315	**3122**
100	6084	4365	2605	3337	4281	4134
Average	6228	4173	2769	**2405**	3523	

Table 3
Performance of MA in comparison with the mixed-integer linear programming model Cuts-C by Fischer et al. [10] on small-size benchmark instances with $n = 25$ facilities (n_1 is the number of facilities in the first row).

Instance	n_1	Cuts-C [10]		MA	
		Optimal value	Time (s)	F	Time (s)
AV25_1	$\lfloor n/2 \rfloor$	2349.0	18046.46	2349.0	0.023
AV25_2	$\lfloor n/2 \rfloor$	19138.5	7039.98	19138.5	0.025
AV25_3	$\lfloor n/2 \rfloor$	12549.0	18714.69	12549.0	0.054
AV25_4	$\lfloor n/2 \rfloor$	24922.5	6972.88	24922.5	0.059
AV25_5	$\lfloor n/2 \rfloor$	8011.0	6683.53	8011.0	0.062
AV25_1	$\lfloor n/3 \rfloor$	3077.0	5980.89	3077.0	0.010
AV25_2	$\lfloor n/3 \rfloor$	23826.5	1240.18	23826.5	0.005
AV25_3	$\lfloor n/3 \rfloor$	18714.0	1152.37	18714.0	0.001
AV25_4	$\lfloor n/3 \rfloor$	30647.5	1673.60	30647.5	0.009
AV25_5	$\lfloor n/3 \rfloor$	10126.0	1616.57	10126.0	0.001
AV25_1	$\lfloor n/4 \rfloor$	3705.0	1183.30	3705.0	0.006
AV25_2	$\lfloor n/4 \rfloor$	26229.5	1413.09	26229.5	0.002
AV25_3	$\lfloor n/4 \rfloor$	23081.0	864.67	23081.0	0.002
AV25_4	$\lfloor n/4 \rfloor$	33584.5	1851.56	33584.5	0.001
AV25_5	$\lfloor n/4 \rfloor$	11289.0	2567.98	11289.0	0.001
AV25_1	$n/5$	4039.0	492.69	4039.0	0.013
AV25_2	$n/5$	30193.5	693.99	30193.5	0.001
AV25_3	$n/5$	23167.0	601.78	23167.0	0.013
AV25_4	$n/5$	38689.5	1009.63	38689.5	0.001
AV25_5	$n/5$	12951.0	843.65	12951.0	0.001

where c is the relative difference, f is the objective value obtained by MA with a certain combination of parameters, and f^* is the lowest f value among all the parameter combinations tested. Using Equation (33), we calculated the relative difference c for each instance in the training dataset. The average values of c over this dataset are reported in Table 2. Its bottom row gives the average results over all values of the parameter M_{pop} and its last column gives the average results over all values of the parameter M_{gen}. The minimum value of the average relative difference is shown in boldface. It was achieved by the MA run with $M_{pop} = 75$ and $M_{gen} = 40$. We used these parameter values for our main experiments.

4.3. Comparison with state-of-the-art results

We start this section by comparing results of our algorithm with those obtained by state-of-the-art exact approaches presented in [10,8]. The aim of the experiment is to show that MA is able to find optimal solutions for small PROP instances in a fraction of a second. Fischer et al. [10] investigated several strategies for solving their MILP formulation of the problem. They have concluded that there was no clear winner among the tested strategies. For comparison purposes, we have selected an implementation called Cuts-C. It found provably optimal solutions for all benchmark instances in the dataset (a). These PROP instances were among the largest ones solved to optimality in [10]. The objective function values of the optimal solutions are displayed in the third column of Table 3. The first two columns of the table show instance name and the number of facilities in the first row. The last two columns report the results of MA, namely, the objective value and running time, both averaged over 20 runs. To make the table more complete, we include running times of Cuts-C. These data, given in the fourth column, are cited from [10]. We can see from the table that MA consistently yielded optimal solutions for all instances in the AV25 set. The table also shows that this was achieved in less than 0.1 seconds per run. Similar conclusions were obtained when running MA on the dataset (b). Instances in this dataset with $n_1 \in \{n/2, n/3\}$ were solved to proven optimality by Yang et al. [8] using their mixed-integer programming model, called PROP2. The results of MA for these instances are shown in the last two columns of Table 4. The results of PROP2 (third and fourth columns) are cited from [8]. From Table 4, we notice that, again, MA was able to find an optimal solution for each instance in the P24 set by spending just a fraction of a second.

Table 4
Performance of MA in comparison with the mixed-integer programming model PROP2 by Yang et al. [8] on small-size benchmark instances with $n = 24$ facilities (n_1 is the number of facilities in the first row).

Instance	n_1	PROP2 [8]		MA	
		Optimal value	Time (s)	F	Time (s)
P24_a	$n/2$	11778.0	17580.62	11778.0	0.028
P24_b	$n/2$	16178.5	19177.99	16178.5	0.017
P24_c	$n/2$	17033.0	17132.04	17033.0	0.018
P24_d	$n/2$	15817.0	10766.44	15817.0	0.026
P24_e	$n/2$	18530.0	27138.77	18530.0	0.011
P24_a	$n/3$	14730.0	2565.00	14730.0	0.001
P24_b	$n/3$	19113.5	26703.12	19113.5	0.002
P24_c	$n/3$	20495.0	4362.23	20495.0	0.002
P24_d	$n/3$	20655.0	7104.45	20655.0	0.002
P24_e	$n/3$	22243.0	11862.98	22243.0	0.007

Table 5
Time limit (in seconds) for instances of size $n \leqslant 70$.

Instance series	n	Time limit (s)			
		$n_1 = \lfloor n/2 \rfloor$	$n_1 = \lfloor n/3 \rfloor$	$n_1 = \lfloor n/4 \rfloor$	$n_1 = \lfloor n/5 \rfloor$
sko56 [86]	56	8	3	11	4
AKV_60 [16]	60	15	4	3	2
AKV_70 [16]	70	15	5	4	3

Table 6
Comparison of MA with the AILS0 (Cravo and Amaral [14]) and QL-PHH (Liu et al. [15]) algorithms (n_1 is the number of facilities in the first row).

Instance	n_1	BKV	AILS0		QL-PHH	MA		
			E_{avg}	SD	SD	E_{avg}	SD	Time (s)
sko56_1	$\lfloor n/2 \rfloor$	32186	0.10	0.3	n/a[a]	0	0	3.5
	$\lfloor n/4 \rfloor$	49443	3.00	6.2	n/a	0	0	3.4
sko56_3	$\lfloor n/2 \rfloor$	85713	10.30	31.7	n/a	0.70	1.0	3.1
sko56_4	$\lfloor n/2 \rfloor$	158686	69.25	49.1	n/a	3.45	11.2	4.4
sko56_5	$\lfloor n/3 \rfloor$	373994.5	1.20	1.0	n/a	0	0	1.6
AKV_60_1	$\lfloor n/5 \rfloor$	1302507	0.65	2.9	0	0	0	0.1
AKV_60_3	$\lfloor n/2 \rfloor$	331103.5	52.80	129.0	227.3	9.25	16.0	7.1
AKV_60_4	$\lfloor n/2 \rfloor$	201052	75.95	80.2	60.6	0	0	1.3
	$\lfloor n/3 \rfloor$	267503	0.30	0.9	0	0	0	2.0
AKV_70_2	$\lfloor n/2 \rfloor$	737919	194.75	486.4	61.0	0	0	6.7
	$\lfloor n/5 \rfloor$	1230230	0	0	32.3	0	0	0.2
AKV_70_3	$\lfloor n/2 \rfloor$	764463.5	594.30	399.2	268.5	0	0	2.7
	$\lfloor n/4 \rfloor$	1163070.5	0.30	1.3	30.7	0	0	1.6
	$\lfloor n/5 \rfloor$	1355113.5	110.85	91.9	0	0	0	0.8
Average			79.55	91.4	75.6	0.96	2.0	2.8

[a] n/a denotes "not available".

Further, we focus on the comparison of MA with metaheuristic algorithms for the PROP. As noted in the introduction, two recent state-of-the-art methods for this problem are adaptive iterated local search (AILS) algorithm by Cravo and Amaral [14] and parallel hyper heuristic algorithm based on reinforcement learning presented by Liu et al. [15]. In this section, we compare the experimental results reported by these authors with those obtained by our algorithm. For instances in the datasets (c)–(e), the MA was run with time limits listed in Table 5. These limits are the same or less than those adopted by Cravo and Amaral (see Table 2 in [14]). To run AILS, Cravo and Amaral used a desktop computer with an Intel Core i7-7700k (4.2 GHz) processor. The single thread rating of their computer is 2727 and that of our PC is 2453 (the performance of CPUs can be compared at https://www.cpubenchmark.net/singleCompare.php). Thus, our computer is not as fast as the computer used in [14].

Table 6 summarizes comparison results between three algorithms. The AILS algorithm of Cravo and Amaral is represented by one of its versions, AILS0. This version performed better than AILS1 and AILS2 on instances with $n \leqslant 70$ (see [14]). In the table, QL-PHH refers to the reinforcement learning-based parallel hyper heuristic algorithm by Liu et al. [15]. It should be noted that all three algorithms succeeded in finding the best known value (BKV given in the third column of the table) for all $60 = 15 \times 4$ PROP instances obtained by using datasets (c)–(e), each of size 5, and setting n_1 to one of the four values: $\lfloor n/m \rfloor$, $m = 2, 3, 4, 5$. Even more, in most cases, all algorithms achieved the best result in all the runs and got the perfect score with zero standard deviation (denoted as SD in the table). There is no sufficient reason to use such test cases for comparison of algorithms. We present results only for those

Table 7
Best results obtained by the AILS1 algorithm (Cravo and Amaral [14]) and MA for instances of size $n \in [250, 300]$ (n_1 is the number of facilities in the first row as provided in Cravo and Amaral [14]).

Instance	n_1	AILS1	MA	
		F_{best}	F_{best}	SR
p250	130	27289929.5	**27289482.5**	4/20
p260	131	31683937.5	**31679712.5**	1/20
p270	130	34552132.5	**34551937.5**	12/20
p280	141	36945712.0	**36945116.0**	5/20
p290	148	43152315.5	**43150530.5**	5/20
p300	158	48012146.0	**48010185.0**	8/20
Average		36939362.2	36937827.3	5.8/20

Table 8
Average results of the AILS1 algorithm (Cravo and Amaral [14]) and MA for instances of size $n \in [250, 300]$ (n_1 is the number of facilities in the first row as provided in Cravo and Amaral [14]).

Instance	n_1	AILS1		MA		
		F_{avg}	SD	F_{avg}	SD	Time (s)
p250	130	27297349.3	5807.9	**27289964.4**	860.3	2203.3
p260	131	31691381.5	8490.3	**31684773.0**	2734.5	2630.7
p270	130	34559840.7	8679.6	**34552510.7**	780.4	1497.4
p280	141	36967634.6	14973.0	**36947240.7**	2471.8	2399.4
p290	148	43169892.2	19203.0	**43151790.7**	2448.1	2122.2
p300	158	48043636.1	14325.8	**48012731.8**	4293.1	2625.1
Average		36954955.7	11913.3	36939835.2	2264.7	2246.3

PROP instances for which at least one of the algorithms failed to provide solutions with zero SD. As it can be seen in Table 6, there are 14 such instances (out of 60) in the testbed. They are identified by their names and the value of the problem parameter n_1 as given in the first two columns in the table. To assess the performance of algorithms, we calculate $E_{avg} = F_{avg} - F_{best}$, where F_{avg} is the objective value averaged over 20 runs of the algorithm and F_{best} is the best known value (given in the third column in Table 6). In the table, we present the values of E_{avg} and SD for AILS0 and MA, and the values of SD for QL-PHH. The results of AILS0 (columns 4 and 5) are extracted from Tables 3–6 in [14]. Liu et al. [15] did not report results for sko56 instances and did not provide the average objective values of solutions found by their algorithm. Therefore, we show in Table 6 only the available SD values obtained by QL-PHH (column 6). These data are extracted from Tables 5–8 in [15]. The last column in Table 6 displays the average time spent by MA until the last improvement in solution quality is made. The last row in the table shows the mean results obtained by the three algorithms.

Inspecting the table, we can see that the MA is superior to the other two algorithms involved in comparison. The MA performed better than AILS0 and QL-PHH in terms of standard deviation (2.0 vs. 91.4 for AILS0 and 75.6 for QL-PHH). It obtained the best solution in all 20 runs for all but three problem instances. An important conclusion from the table is that the benchmark PROP instances in the datasets (c)–(e) seem to be quite easy for all of the state-of-the-art algorithms.

For more thorough testing, we ran the MA on problem instances in the dataset (f) with $250 \leqslant n \leqslant 500$. We set the maximum time limit to 3600 seconds per run. The same time limit was used by Cravo and Amaral [14] for their AILS algorithm. We compare the results of the MA with those obtained by AILS1. This configuration of AILS strongly outperformed the other two configurations, AILS0 and AILS2, for large-scale problem instances (see Table 8 in [14]).

The performance results of algorithms are compared in Tables 7 and 8. The third and fourth columns of Table 7 show the best objective value, F_{best}, found in 20 independent runs of AILS1 and MA, respectively. The last column presents the success rate, SR, for MA, defined as the fraction of runs in which the best result was achieved. In Table 8, the columns under heading F_{avg} report the objective value averaged over 20 runs. The columns labeled SD show the standard deviation of the F values. The last column displays the average CPU time required by MA to get the best result in a run. The bottom row of each table gives the averaged results. The better of the two F_{best} values (in Table 7) and F_{avg} values (in Table 8) for every instance are marked in bold font.

We see from Tables 7 and 8 that our algorithm performs well on larger PROP instances. It found better solutions than AILS1 for all 6 instances in the test suite. As seen from the last column in Table 7, for all instances except p260, the MA obtained the best result in 4 or more runs. Comparing average values (Table 8), we observe a clear dominance of the MA over the AILS1 algorithm. The MA yielded better results in terms of F_{avg} for all instances used in the experiment. We also see that the standard deviation values obtained by MA are about 5 times smaller on average than those obtained by AILS1. Summarizing, our algorithm can be considered as an attractive alternative to state-of-the-art heuristic techniques for the PROP.

Table 9
Performance comparison of MA and MA′ on large PROP instances (n_1 is the number of facilities in the first row).

Instance	n_1	MA			MA′		
		F_{best}	F_{avg}	SR	F_{best}	F_{avg}	SR
p250	130	**27289482.5**	*27289964.4*	4/20	**27289482.5**	27293920.8	2/20
p260	131	**31679712.5**	*31684773.0*	1/20	31680182.5	31689237.9	1/20
p270	130	**34551937.5**	*34552510.7*	12/20	**34551937.5**	34552835.2	8/20
p280	141	**36945116.0**	*36947240.7*	5/20	**36945116.0**	36951546.3	2/20
p290	148	**43150530.5**	*43151790.7*	5/20	43150548.5	43154712.2	3/20
p300	158	**48010185.0**	*48012731.8*	8/20	**48010185.0**	48016508.5	7/20
p310	159	**52910101.0**	*52913776.9*	4/20	**52910101.0**	52918639.0	2/20
p320	163	**59795803.5**	*59800730.4*	5/20	**59795803.5**	59802177.2	1/20
p330	169	**62523654.5**	62544749.1	1/20	62523989.5	*62542169.7*	1/20
p340	165	**64939505.5**	*64943906.7*	4/20	**64939505.5**	64946954.0	2/20
p350	162	**74994312.0**	*74996029.4*	10/20	**74994312.0**	74996502.9	5/20
p360	183	70624507.0	70641479.4	1/20	**70624472.0**	*70634230.3*	1/20
p370	178	87404510.5	*87415512.1*	1/20	**87404384.5**	87419358.3	2/20
p380	189	94836376.5	94869448.3	1/20	**94833486.5**	*94859158.1*	2/20
p390	188	104457977.0	104466446.7	1/20	**104457930.0**	*104464515.7*	2/20
p400	197	106978446.5	107020501.9	1/20	**106963216.5**	*106992121.8*	1/20
p410	213	121951122.0	121989836.2	1/20	**121945093.0**	*121967991.5*	1/20
p420	210	135561596.5	135587834.3	1/20	**135554982.5**	*135568447.9*	1/20
p430	213	143287068.5	143342796.1	1/20	**143280378.5**	*143314727.8*	1/20
p440	219	150633888.5	150683338.2	1/20	**150631245.5**	*150645700.4*	1/20
p450	221	162328305.0	162384762.5	1/20	**162318129.0**	*162349590.4*	1/20
p460	234	157532037.0	157621693.1	1/20	**157514205.0**	*157567486.8*	1/20
p470	232	189919778.0	189968582.7	1/20	**189858413.0**	*189894177.4*	1/20
p480	248	183696505.0	183758760.5	1/20	**183628819.0**	*183698008.6*	1/20
p490	246	207086478.5	207179792.2	1/20	**207040927.5**	*207100582.4*	1/20
p500	241	232978593.5	233059515.9	1/20	**232928659.5**	*233005195.8*	1/20
Average		103310289.6	103339557.8	2.8/20	103299057.9	103321019.1	2.0/20

4.4. Experiments on large-scale PROP instances

This section presents the results of testing the performance of our algorithm on PROP instances of size up to $n = 500$. We compared two versions of the algorithm, MA and MA′. We remind that the baseline version MA evaluates individuals of a population using a fitness function that balances the objective value with the diversity of population. The MA′ version forms the next population simply by taking the best M_{pop} solutions from the pool. As before, we set a cutoff time of 3600 seconds per run of our algorithm.

Table 9 compares the performance of the two versions of the memetic algorithm. The first two columns of this table indicate the PROP instances used in the comparison. The digits in their name represent the total number of facilities. The number of facilities in the first row of the layout is displayed in the second column. This number, n_1, is set to the value of $i \in \{1, \ldots, n\}$ minimizing $|\Lambda_i - \bar{\Lambda}_i|$, where Λ_i is the sum of lengths of the first i facilities in the PROP instance and $\bar{\Lambda}_i$ is the sum of lengths of the remaining $n - i$ facilities in the same instance. The idea is to make the lengths of both rows close each other. The next columns show the best and average objective function values, F_{best} and F_{avg}, and the success rate of each version of the algorithm in 20 runs. The better of the two F_{best} values and the better of the two F_{avg} values for each instance is shown in boldface and in italics, respectively. The last row of the table indicates the average value of each column. The results of the MA for the first 6 instances are collected from Tables 7 and 8.

To evaluate the results, we can split the instances of Table 9 into two groups: those with $n \leqslant 350$ (the first 11 rows of the table) and those with $n \geqslant 360$ (the remaining 15 rows). When restricting the comparison to instances in the first group, the MA version performed better than MA′. It obtained smaller F_{avg} values than MA′ for 10 of 11 instances. However, the picture is different when comparing the two versions of the algorithm for instances in the second group. In this case, the results show a clear superiority of MA′ over MA. The MA′ version found better solutions for all 15 problem instances in the group and achieved smaller average objective function values for 14 instances. One can also observe from Table 9 that, for many instances, both MA and MA′ obtained the best result in only one run (in such cases, SR = 1/20). This shows that the PROP is a hard combinatorial optimization problem.

For the purpose of investigating the reasons why MA and MA′ perform differently for smaller and larger PROP sizes, we calculated the average objective function value of each population generated by these algorithms. As before, the cutoff time for a run was one hour. Fig. 7 presents the results of the experiment for $n = 250, 300, 400$, and 500. We denote by g the generation number and by $F_{pop\_avg}(\text{MA}, g)$ the average objective function value of the population at generation g achieved by the MA (we will use this notation also for MA′). In the figure, g and F_{pop_avg} are shown on the x- and y-axis, respectively. Parts (a) and (b) of the figure provide results for $g \leqslant 200$ only. For larger g, the value of F_{pop_avg} does not change (for MA′) of fluctuate over time (for MA). Specifically, $F_{pop\_avg}(\text{MA}', g) = 27294790.4$ for $g = 79, \ldots, 1278$ in the case of p250 and $F_{pop\_avg}(\text{MA}', g) = 48019776.4$ for $g = 48, \ldots, 1462$ in the case of p300. We note that the number of generations produced by MA in most cases is smaller than that produced by MA′ (407 vs. 1278, 333 vs. 1462, and 20 vs. 24 for p250, p300, and p400, respectively). The main reason is that MA tends to generate slightly worse

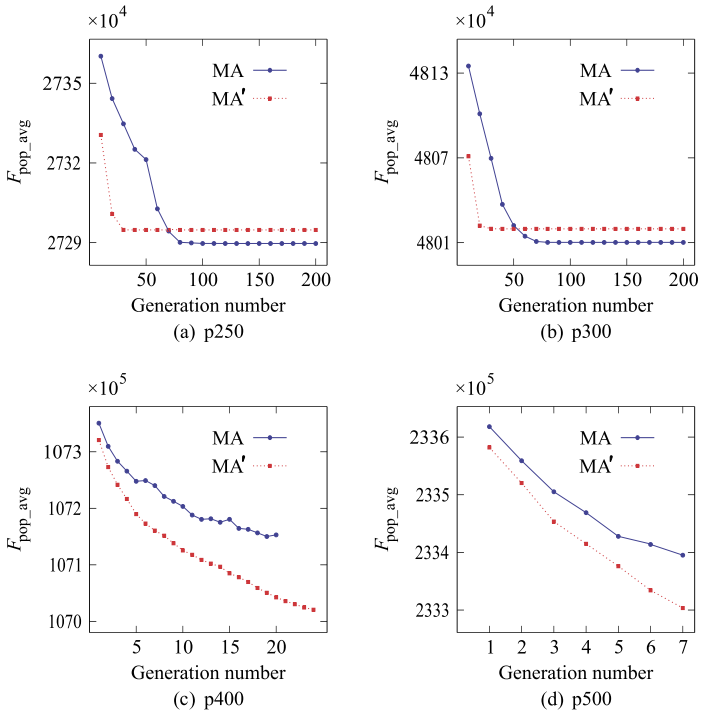

Fig. 7. Relationship between the average objective value of the population and generation number.

offspring than MA′ and, consequently, spends more CPU time when applying the `local_search` procedure (given in Algorithm 1) to this offspring.

As seen in Fig. 7, MA and MA′ behave differently depending on the problem size. Naturally, $F_{\text{pop\_avg}}(\text{MA}', g) < F_{\text{pop\_avg}}(\text{MA}, g)$ at the first generations. However, for smaller PROP instances, there exists a generation, say $\tilde{g}$, such that $F_{\text{pop\_avg}}(\text{MA}, g) \leqslant F_{\text{pop\_avg}}(\text{MA}', g)$ for all $g \geqslant \tilde{g}$. In parts (a) and (b) of the figure, $\tilde{g} = 69$ and $\tilde{g} = 54$, respectively. Thus, the evolution of the population in the case of MA′ falls into stagnation, and this is the reason behind the inferior performance of MA′ for instances `p250` and `p300`. In contrast, the MA version avoids the loss of diversity of the population and therefore prevents the search process from getting trapped at local minima. However, the picture is different for larger PROP instances (parts (c) and (d) of Fig. 7). We see that $F_{\text{pop\_avg}}(\text{MA}', g) < F_{\text{pop\_avg}}(\text{MA}, g)$ for all generations g. At each iteration, the MA′ constructs a population by selecting individuals with the best fitness values. The declining curves for MA′ in Fig. 7 (c) and (d) indicate that such a strategy does not hinder the algorithm from updating (and improving) the population at each generation. Maintaining the elite population with the best individuals increases the chances of obtaining a high-quality solution in reasonable computation time. The MA′ results in Table 9 support this argument.

4.5. Effectiveness of local search

A straightforward implementation of insertion neighborhood exploration in LS algorithms for the PROP has $O(n^4)$ complexity, where n is the number of facilities as before. This fact follows directly from the observation that both the size of the neighborhood and the number of operations required to compute $F(p_1, p_2)$ are proportional to n^2. Recently, Herrán et al. [74] proposed a more efficient approach. Their insertion neighborhood exploration procedure for the space-free multi-row facility layout problem runs in $O(n^3)$ time (see Equation (4) and Figs. 3 and 4 in the Supplementary material of [74]). The method of Herrán et al. [74] can also be used in algorithms for solving the PROP. By virtue of Theorem 1, our technique is superior to that presented in [74]. To complement our analysis, we also compared both techniques experimentally. We modified our MA implementation by replacing the `explore_neighborhood` procedure in Algorithm 1 with the neighborhood examination mechanism proposed in [74]. Certainly, we also removed the initialization statements from Algorithm 1 (Lines 1–14 in the pseudocode). The other parts of MA were left intact. We denote the resulting version of MA as MA-ALS (Memetic Algorithm with Alternative Local Search procedure). Both procedures, that is, Algorithm 1 and alternative LS scan the neighborhood in precisely the same order. Therefore, given the same pair of initial permutations, both LS algorithms arive at the same locally optimal solution.

In order to show the performance difference between MA and MA-ALS, we compare these algorithms on `sko56`, `AKV_60`, and `AKV_70` datasets. For all instances, we fixed n_1 at $n/2$. By performing preliminary experiments, it was found that the time limits of Table 5 were too tight for MA-ALS. Basically, in many cases, the allotted time expired before the construction of initial population was completed. To cope with this issue, we let MA-ALS run longer: 35 s for `sko56`, 40 s for `AKV_60`, and 80 s for `AKV_70`. The comparison results are summarized in Table 10. Each column heading of the table has been defined while presenting Tables 7–9. The minimum value of F_{best} appears in boldface. We can see that MA-ALS failed to obtain the best solution for 5 instances in the

Table 10
Performance comparison of MA and MA-ALS on small-size PROP instances (the number of facilities in the first row is equal to $n/2$).

Instance	MA			MA-ALS		
	F_{best}	F_{avg}	Time (s)	F_{best}	F_{avg}	Time (s)
sko56_1	**32186.0**	32186.0	3.5	32195.0	32215.5	26.9
sko56_2	**259349.0**	259349.0	1.5	**259349.0**	259363.2	25.0
sko56_3	**85713.0**	85713.7	3.1	**85713.0**	85769.6	20.2
sko56_4	**158686.0**	158689.4	4.4	158734.0	158907.1	24.4
sko56_5	**298813.5**	298813.5	4.0	**298813.5**	298911.0	24.4
AKV_60_1	**772202.0**	772202.0	2.1	**772202.0**	772409.0	26.6
AKV_60_2	**430380.0**	430380.0	1.4	**430380.0**	430384.7	22.0
AKV_60_3	**331103.5**	331112.7	7.1	331140.5	331614.9	29.1
AKV_60_4	**201052.0**	201052.0	1.3	**201052.0**	201092.9	27.3
AKV_60_5	**165096.0**	165096.0	3.9	**165096.0**	165131.6	27.4
AKV_70_1	**779130.0**	779130.0	3.8	**779130.0**	779696.3	49.8
AKV_70_2	**737919.0**	737919.0	6.7	737938.0	739177.4	63.1
AKV_70_3	**764463.5**	764463.5	2.7	**764463.5**	764731.1	57.2
AKV_70_4	**491028.0**	491028.0	2.5	**491028.0**	491055.6	53.1
AKV_70_5	**2187780.5**	2187780.5	4.1	2187865.5	2191926.7	57.5
Average	512993.5	512994.4	3.5	513006.7	513492.4	35.6

Table 11
Time (in seconds) taken by MA and MA-ALS to construct the initial population (n_1 is the number of facilities in the first row).

Instance	n_1	MA time (s)	MA-ALS time (s)
p250	130	78.5	9499.5
p260	131	88.8	11191.6
p270	130	101.1	13223.5
p280	141	111.2	15082.4
p290	148	126.3	17738.2
p300	158	141.1	20551.8
Average		107.8	14547.8

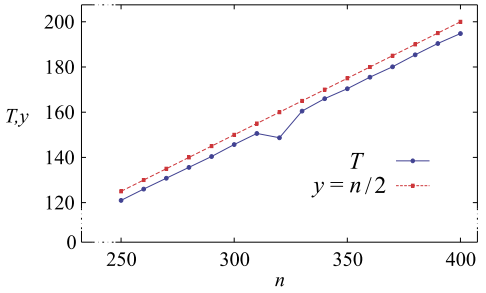

Fig. 8. Dependence of the ratio $T = T_{MA-ALS}/T_{MA}$ on the number of facilities n (the line $y = n/2$ is shown for comparison).

dataset. Comparing F_{avg} and time values of the two approaches, we find that MA dominates MA-ALS across all instances in the table. In particular, it is seen that MA-ALS took an order of magnitude longer to run than MA. Thus, MA has a clear solution quality and CPU time advantage over MA-ALS. Similar comparison results were observed for other values of the problem parameter n_1.

As an additional experiment, we attempted to run MA-ALS on PROP instances of size $n \geqslant 250$. However, the chosen time limit of one hour was insufficient even to build the initial population. Obviously, the best solution obtained at this step of the algorithm cannot be acceptable. Because of inability to get satisfactory solutions using MA-ALS, our analysis was restricted to comparing the computational times required to construct the initial population by each of the MA versions. The results for instances with $n \leqslant 300$ are shown in Table 11. It can be seen that MA-ALS runs two orders of magnitude slower than MA. Letting T_{MA} and T_{MA-ALS} denote the time taken by MA and MA-ALS, respectively, we find that the average value of the ratio $T = T_{MA-ALS}/T_{MA}$ exceeds 130. Fig. 8 displays T as a function of n over a wider range of n values. We can see that the ratio T pretty closely follows the line $y = n/2$. A general conclusion from the experiments is that the local search technique from [74] is too slow to be applied in memetic algorithms.

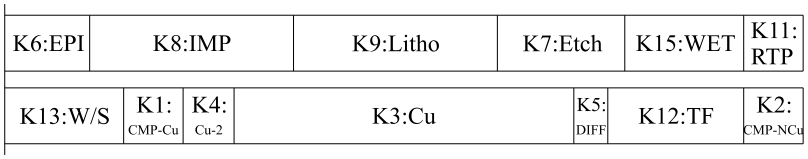

Fig. 9. Solution obtained by MA for HY_13.

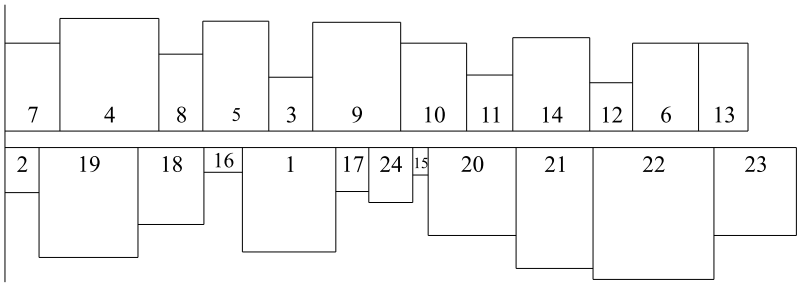

Fig. 10. Solution obtained by MA for IPF24.

4.6. Application examples

As outlined in the introduction, the PROP began to receive considerable attention in recent years only. Therefore, the current literature lacks real-world PROP instances from industry. Considering this fact, we made an attempt to adopt instances of related row layout problems for the PROP. An instance of the multi-row facility layout problem in semiconductor fabrication was presented in [73]. This instance has 13 facilities representing manufacturing processes in semiconductor fabrication in China. In [73], the facilities (processing modules) are denoted as K1–K9, K11–K13, and K15. An instance of the PROP is obtained by specifying a group of facilities to be placed in the first row of the layout. We have constructed such a group quite loosely. We included into the group several closely related processes such as etching (K7: Etch), photolithography (K9: Litho), wet clean (K15: WET), and ion implantation (K8: IMP). We enlarged this group by adding epitaxy (K6: EPI) and rapid thermal processing (K11: RTP). Other processing modules (K1: CMP-Cu, K2: CMP-NCu, K3: Cu, K4: Cu-2, K5: DIFF, K12: TF, and K13: W/S) were assigned to the second row. Let us denote the constructed PROP instance as HY_13. We applied MA to this instance. The obtained layout with the corresponding objective function value of 10220 is shown in Fig. 9. The algorithm took less than a second to produce this result.

Liu et al. [48] presented a real-world instance of the double-floor corridor allocation problem in the fabrication of infant feeding bottles and other plastic products for babies. This instance, named IPF24 in [48], consists of 24 facilities (production areas) of different sizes. The data specifying IPF24 are given in Appendices B1 and B2 of [48]. To construct a PROP instance from IPF24, we formed a group of facilities with similar functions. First, we put all moulding (facilities 3–10) and printing (facilities 11–13) areas into one group. Then, we enlarged this group by adding high temperature cooling area (facility 14). These selected facilities were placed in the first row and the remaining facilities were placed in the second row. The layout produced by our algorithm is depicted in Fig. 10. The objective value of this solution is 677128.5. The solution time taken by the MA is less than a second.

5. Conclusions and future work

In this paper, we solved an open problem regarding the complexity of local search for the PROP by providing an $O(n^2)$-time procedure for insertion neighborhood exploration. The existence of such a procedure allows us to state that the computational complexity of this neighborhood exploration problem is $\Theta(n^2)$. We incorporated the developed procedure in a memetic algorithm for solving the PROP. It is an effective strategy to employ a fast LS heuristic in order to increase the performance of this algorithm.

We have tested the MA using several sets of PROP instances. The algorithm obtained high-quality solutions in a reasonable amount of computer time. The experimental results demonstrate a superiority of the MA over the adaptive ILS algorithm of Cravo and Amaral [14] and parallel hyper heuristic algorithm of Liu et al. [15], which are state-of-the-art for the PROP. The MA found better solutions than those reported by Cravo and Amaral for six largest instances in their experiments.

We also tested the MA on an additional set of PROP instances ranging from 310 to 500 facilities. For many of them, the MA obtained the best result in a single run out of 20 independent runs. So, it can be supposed that the best solutions obtained by MA for large-scale test problems in our experiment are not globally best. Using MA for finding better solutions may require high computational resources. A promising avenue for future investigations involves developing innovative fast heuristic algorithms capable of obtaining solutions of high quality for large-scale PROP instances. One possibility in this respect would be to adopt recent evolutionary computing techniques for solving the PROP. Along this line of research, the proposed LS algorithm can be employed as an effective mechanism for search intensification.

An intriguing direction for future work is extending the proposed LS technique to the case of so-called k-PROP [7], which is a generalization of the PROP. The k-PROP assumes that the partition of the set of facilities into $k \geqslant 2$ groups is given explicitly in advance and, moreover, each group is assigned a separate row in the layout. The problem asks to place facilities of each group in their corresponding row. For $k > 2$, the objective function is a straightforward generalization of (1). Using fast LS, it would be possible to develop effective evolutionary algorithms like MA for solving the k-PROP.

Another well-known type of neighborhood for permutation problems is an interchange (or 2-opt) neighborhood. It consists of all permutations that can be generated from the given base permutation by swapping positions of two elements in the base permutation. There are algorithms for FLPs which use insertion and interchange neighborhoods in a combined way. Examples are LS procedures used in the VNS algorithms for the dynamic SRFLP [39] and for the MRFLP [74]. The performance of heuristic algorithms for FLPs can be enhanced when efficient strategies for interchange neighborhood exploration are available. However, in the case of the PROP, determining the complexity of the interchange neighborhood exploration problem is an open question. We conjecture that the computational complexity of this problem is $\Theta(n^2)$. To settle this conjecture, an $O(n^2)$-time algorithm for exploring the interchange neighborhood needs to be constructed.

Data availability

Data will be made available on request.

References

[1] S.S. Heragu, Facilities Design, fourth ed., CRC Press, Boca Raton, FL, 2016.
[2] A. Drira, H. Pierreval, S. Hajri-Gabouj, Facility layout problems: a survey, Annu. Rev. Control 31 (2007) 255–267.
[3] J.A. Tompkins, J.A. White, Y.A. Bozer, J.M.A. Tanchoco, Facilities Planning, fourth ed., John Wiley & Sons, 2010.
[4] M.F. Anjos, M.V.C. Vieira, Mathematical optimization approaches for facility layout problems: the state-of-the-art and future research directions, Eur. J. Oper. Res. 261 (2017) 1–16.
[5] H. Hosseini-Nasab, S. Fereidouni, S.M.T. Fatemi Ghomi, M.B. Fakhrzad, Classification of facility layout problems: a review study, Int. J. Adv. Manuf. Technol. 94 (2018) 957–977.
[6] B. Keller, U. Buscher, Single row layout models, Eur. J. Oper. Res. 245 (2015) 629–644.
[7] A.R.S. Amaral, A parallel ordering problem in facilities layout, Comput. Oper. Res. 40 (2013) 2930–2939.
[8] X. Yang, W. Cheng, A.E. Smith, A.R.S. Amaral, An improved model for the parallel row ordering problem, J. Oper. Res. Soc. 71 (2020) 475–490.
[9] J. Gong, Z. Zhang, J. Liu, C. Guan, S. Liu, Hybrid algorithm of harmony search for dynamic parallel row ordering problem, J. Manuf. Syst. 58 (2021) 159–175.
[10] A. Fischer, F. Fischer, P. Hungerländer, New exact approaches to row layout problems, Math. Program. Comput. 11 (2019) 703–754.
[11] P. Hungerländer, M.F. Anjos, A semidefinite optimization-based approach for global optimization of multi-row facility layout, Eur. J. Oper. Res. 245 (2015) 46–61.
[12] M. Dahlbeck, A. Fischer, F. Fischer, Decorous combinatorial lower bounds for row layout problems, Eur. J. Oper. Res. 286 (2020) 929–944.
[13] M. Maadi, M. Javidnia, R. Jamshidi, Two strategies based on meta-heuristic algorithms for parallel row ordering problem (PROP), Iran. J. Manag. Stud. 10 (2017) 467–498.
[14] G.L. Cravo, A.R.S. Amaral, Adaptive iterated local search for the parallel row ordering problem, Expert Syst. Appl. 208 (2022) 118033, https://doi.org/10.1016/j.eswa.2022.118033.
[15] J. Liu, Z. Zhang, S. Liu, Y. Zhang, T. Wu, Parallel hyper heuristic algorithm based on reinforcement learning for the corridor allocation problem and parallel row ordering problem, Adv. Eng. Inform. 56 (2023) 101977, https://doi.org/10.1016/j.aei.2023.101977.
[16] M.F. Anjos, A. Kennings, A. Vannelli, A semidefinite optimization approach for the single-row layout problem with unequal dimensions, Discrete Optim. 2 (2005) 113–122.
[17] M.F. Anjos, A. Vannelli, Computing globally optimal solutions for single-row layout problems using semidefinite programming and cutting planes, INFORMS J. Comput. 20 (2008) 611–617.
[18] P. Hungerländer, F. Rendl, A computational study and survey of methods for the single-row facility layout problem, Comput. Optim. Appl. 55 (2013) 1–20.
[19] A.R.S. Amaral, A new lower bound for the single row facility layout problem, Discrete Appl. Math. 157 (2009) 183–190.
[20] A.R.S. Amaral, On the exact solution of a facility layout problem, Eur. J. Oper. Res. 173 (2006) 508–518.
[21] A.R.S. Amaral, An exact approach to the one-dimensional facility layout problem, Oper. Res. 56 (2008) 1026–1033.
[22] A.R.S. Amaral, A.N. Letchford, A polyhedral approach to the single row facility layout problem, Math. Program. 141 (2013) 453–477.
[23] K. Maier, V. Taferner, Solving the constrained single-row facility layout problem with integer linear programming, Int. J. Prod. Res. 61 (2023) 1882–1897.
[24] D. Datta, A.R.S. Amaral, J.R. Figueira, Single row facility layout problem using a permutation-based genetic algorithm, Eur. J. Oper. Res. 213 (2011) 388–394.
[25] F. Ozcelik, A hybrid genetic algorithm for the single row layout problem, Int. J. Prod. Res. 50 (2012) 5872–5886.
[26] R. Kothari, D. Ghosh, An efficient genetic algorithm for single row facility layout, Optim. Lett. 8 (2014) 679–690.
[27] R. Kothari, D. Ghosh, Tabu search for the single row facility layout problem using exhaustive 2-opt and insertion neighborhoods, Eur. J. Oper. Res. 224 (2013) 93–100.
[28] X. Ning, P. Li, A cross-entropy approach to the single row facility layout problem, Int. J. Prod. Res. 56 (2018) 3781–3794.
[29] G. Palubeckis, Fast local search for single row facility layout, Eur. J. Oper. Res. 246 (2015) 800–814.
[30] J. Guan, G. Lin, Hybridizing variable neighborhood search with ant colony optimization for solving the single row facility layout problem, Eur. J. Oper. Res. 248 (2016) 899–909.
[31] M. Rubio-Sánchez, M. Gallego, F. Gortázar, A. Duarte, GRASP with path relinking for the single row facility layout problem, Knowl. Based Syst. 106 (2016) 1–13.
[32] R. Kothari, D. Ghosh, A scatter search algorithm for the single row facility layout problem, J. Heuristics 20 (2014) 125–142.
[33] G. Palubeckis, Single row facility layout using multi-start simulated annealing, Comput. Ind. Eng. 103 (2017) 1–16.
[34] G.L. Cravo, A.R.S. Amaral, A GRASP algorithm for solving large-scale single row facility layout problems, Comput. Oper. Res. 106 (2019) 49–61.
[35] S. Liu, Z. Zhang, C. Guan, L. Zhu, M. Zhang, P. Guo, An improved fireworks algorithm for the constrained single-row facility layout problem, Int. J. Prod. Res. 59 (2021) 2309–2327.
[36] L. Tang, Z. Li, J.-K. Hao, Solving the single-row facility layout problem by K-medoids memetic permutation group, IEEE Trans. Evol. Comput. 27 (2023) 251–265, https://doi.org/10.1109/TEVC.2022.3165987.
[37] X. Sun, P. Chou, C.-S. Koong, C.-C. Wu, L.-R. Chen, Optimizing 2-opt-based heuristics on GPU for solving the single-row facility layout problem, Future Gener. Comput. Syst. 126 (2022) 91–109.

[38] R. Şahin, S. Niroomand, E.D. Durmaz, S. Molla-Alizadeh-Zavardehi, Mathematical formulation and hybrid meta-heuristic solution approaches for dynamic single row facility layout problem, Ann. Oper. Res. 295 (2020) 313–336.
[39] G. Palubeckis, A. Ostreika, J. Platužienė, A variable neighborhood search approach for the dynamic single row facility layout problem, Mathematics 10 (2022) 2174, https://doi.org/10.3390/math10132174.
[40] A.R.S. Amaral, The corridor allocation problem, Comput. Oper. Res. 39 (2012) 3325–3330.
[41] H. Ahonen, A.G. de Alvarenga, A.R.S. Amaral, Simulated annealing and tabu search approaches for the corridor allocation problem, Eur. J. Oper. Res. 232 (2014) 221–233.
[42] Z. Kalita, D. Datta, Solving the bi-objective corridor allocation problem using a permutation-based genetic algorithm, Comput. Oper. Res. 52 (2014) 123–134.
[43] Z. Kalita, D. Datta, G. Palubeckis, Bi-objective corridor allocation problem using a permutation-based genetic algorithm hybridized with a local search technique, Soft Comput. 23 (2019) 961–986.
[44] C. Guan, Z. Zhang, Y. Li, A flower pollination algorithm for the double-floor corridor allocation problem, Int. J. Prod. Res. 57 (2019) 6506–6527.
[45] Z. Zhang, L. Mao, C. Guan, L. Zhu, Y. Wang, An improved scatter search algorithm for the corridor allocation problem considering corridor width, Soft Comput. 24 (2020) 461–481.
[46] C. Guan, Z. Zhang, J. Gong, S. Liu, Mixed integer linear programming model and an effective algorithm for the bi-objective double-floor corridor allocation problem, Comput. Oper. Res. 132 (2021) 105283.
[47] S. Liu, Z. Zhang, C. Guan, J. Liu, R. Dewil, Mathematical formulation and a new metaheuristic for the constrained double-floor corridor allocation problem, J. Manuf. Syst. 61 (2021) 155–170.
[48] J. Liu, Z. Zhang, Y. Zhang, S. Liu, F. Chen, T. Yin, Mixed-integer programming model and hybrid immune clone select algorithm for multi-objective double floor corridor allocation problem with vertical conveyor, Robot. Comput.-Integr. Manuf. 77 (2022) 102364, https://doi.org/10.1016/j.rcim.2022.102364.
[49] J. Liu, Z. Zhang, F. Chen, S. Liu, L. Zhu, A novel hybrid immune clonal selection algorithm for the constrained corridor allocation problem, J. Intell. Manuf. 33 (2022) 953–972.
[50] J. Liu, Z. Zhang, J. Gong, F. Chen, T. Yin, Y. Zhang, A novel hybrid clonal selection algorithm for the corridor allocation problem with irregular material handling positions, Comput. Ind. Eng. 168 (2022) 108118.
[51] Z. Zhang, J. Gong, J. Liu, F. Chen, A fast two-stage hybrid meta-heuristic algorithm for robust corridor allocation problem, Adv. Eng. Inform. 53 (2022) 101700.
[52] E.D. Durmaz, R. Şahin, An efficient iterated local search algorithm for the corridor allocation problem, Expert Syst. Appl. 212 (2023) 118804.
[53] R.F.R. Correa, H.S. Bernardino, J.M. de Freitas, S.S.R.F. Soares, L.B. Gonçalves, L.L.O. Moreno, A grammar-based genetic programming hyper-heuristic for corridor allocation problem, in: J.C. Xavier-Junior, R.A. Rios (Eds.), Intelligent Systems (BRACIS 2022), in: Lecture Notes in Computer Science, vol. 13653, Springer, Cham, 2022, pp. 504–519.
[54] J. Chung, J.M.A. Tanchoco, The double row layout problem, Int. J. Prod. Res. 48 (2010) 709–727.
[55] A.R.S. Amaral, Optimal solutions for the double row layout problem, Optim. Lett. 7 (2013) 407–413.
[56] A.R.S. Amaral, A mixed-integer programming formulation for the double row layout of machines in manufacturing systems, Int. J. Prod. Res. 57 (2019) 34–47.
[57] L.D. Secchin, A.R.S. Amaral, An improved mixed-integer programming model for the double row layout of facilities, Optim. Lett. 13 (2019) 193–199.
[58] J. Chae, A.C. Regan, A mixed integer programming model for a double row layout problem, Comput. Ind. Eng. 140 (2020) 106244.
[59] A.R.S. Amaral, A mixed-integer programming formulation of the double row layout problem based on a linear extension of a partial order, Optim. Lett. 15 (2021) 1407–1423.
[60] A.R.S. Amaral, A mixed-integer programming formulation for optimizing the double row layout problem, Optim. Methods Softw., https://doi.org/10.1080/10556788.2024.2349093.
[61] C.C. Murray, A.E. Smith, Z. Zhang, An efficient local search heuristic for the double row layout problem with asymmetric material flow, Int. J. Prod. Res. 51 (2013) 6129–6139.
[62] J. Guan, G. Lin, H.-B. Feng, Z.-Q. Ruan, A decomposition-based algorithm for the double row layout problem, Appl. Math. Model. 77 (2020) 963–979.
[63] A.P. Rifai, S.T. Windras Mara, H. Ridho, R. Norcahyo, The double row layout problem with safety consideration: a two-stage variable neighborhood search approach, J. Ind. Prod. Eng. 39 (2022) 181–195.
[64] A.R.S. Amaral, A heuristic approach for the double row layout problem, Ann. Oper. Res. 316 (2022) 837–872.
[65] S. Liu, Z. Zhang, C. Guan, J. Liu, J. Gong, R. Dewil, Mathematical formulation and two-phase optimisation methodology for the constrained double-row layout problem, Neural Comput. Appl. 34 (2022) 6907–6926.
[66] D. Ji, Z. Zhang, W. Liang, C. Wang, Z. He, Mathematical formulation and a novel two-stage algorithm for double-row layout problem with fixed loading and unloading points, J. Manuf. Syst. 69 (2023) 242–254.
[67] X.Q. Zuo, C.C. Murray, A.E. Smith, Sharing clearances to improve machine layout, Int. J. Prod. Res. 54 (2016) 4272–4285.
[68] S. Wang, X. Zuo, X. Liu, X. Zhao, J. Li, Solving dynamic double row layout problem via combining simulated annealing and mathematical programming, Appl. Soft Comput. 37 (2015) 303–310.
[69] M.F. Anjos, M.V.C. Vieira, Mathematical optimization approach for facility layout on several rows, Optim. Lett. 15 (2021) 9–23.
[70] M.F. Anjos, A. Fischer, P. Hungerländer, Improved exact approaches for row layout problems with departments of equal length, Eur. J. Oper. Res. 270 (2018) 514–529.
[71] A. Tubaileh, J. Siam, Single and multi-row layout design for flexible manufacturing systems, Int. J. Comput. Integr. Manuf. 30 (2017) 1316–1330.
[72] S. Safarzadeh, H. Koosha, Solving an extended multi-row facility layout problem with fuzzy clearances using GA, Appl. Soft Comput. 61 (2017) 819–831.
[73] B. Hu, B. Yang, A particle swarm optimization algorithm for multi-row facility layout problem in semiconductor fabrication, J. Ambient Intell. Humaniz. Comput. 10 (2019) 3201–3210.
[74] A. Herrán, J.M. Colmenar, A. Duarte, An efficient variable neighborhood search for the space-free multi-row facility layout problem, Eur. J. Oper. Res. 295 (2021) 893–907.
[75] Y. Li, Z. Li, Bi-objective optimization for multi-row facility layout problem integrating automated guided vehicle path, IEEE Access 11 (2023) 55954–55964, https://doi.org/10.1109/ACCESS.2023.3281554.
[76] X. Zuo, S. Gao, M.-C. Zhou, X. Yang, X. Zhao, A three-stage approach to a multirow parallel machine layout problem, IEEE Trans. Autom. Sci. Eng. 16 (2019) 433–447.
[77] X. Wan, X. Zuo, X. Li, X. Zhao, A hybrid multiobjective GRASP for a multi-row facility layout problem with extra clearances, Int. J. Prod. Res. 60 (2022) 957–976.
[78] C. Guan, Z. Zhang, L. Zhu, S. Liu, Mathematical formulation and a hybrid evolution algorithm for solving an extended row facility layout problem of a dynamic manufacturing system, Robot. Comput.-Integr. Manuf. 78 (2022) 102379, https://doi.org/10.1016/j.rcim.2022.102379.
[79] N.R. Uribe, A. Herrán, J.M. Colmenar, A. Duarte, An improved GRASP method for the multiple row equal facility layout problem, Expert Syst. Appl. 182 (2021) 115184.
[80] A.C. Nearchou, Meta-heuristics from nature for the loop layout design problem, Int. J. Prod. Econ. 101 (2006) 312–328.
[81] G. Palubeckis, An approach integrating simulated annealing and variable neighborhood search for the bidirectional loop layout problem, Mathematics 9 (2021) 5, https://doi.org/10.3390/math9010005.
[82] W. Sun, J.-K. Hao, W. Li, Q. Wu, An adaptive memetic algorithm for the bidirectional loop layout problem, Knowl. Based Syst. 258 (2022) 110002.
[83] M.F. Anjos, M.V.C. Vieira, Facility Layout, EURO Advanced Tutorials on Operational Research, Springer Nature, Switzerland, 2021.

[84] G. Syswerda, Schedule optimization using genetic algorithms, in: L. Davis (Ed.), Handbook of Genetic Algorithms, Van Nostrand Reinhold, New York, 1991, pp. 332–349.
[85] R. Kumar, M. Memoria, M. Thapliyal, M. Kirola, I. Ahmad, A. Gupta, S. Tyagi, N. Ansari, Analyzing the performance of crossover operators (OX, OBX, PBX, MPX) to solve combinatorial problems, in: 2022 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COM-IT-CON), IEEE, Faridabad, India, 2022, pp. 817–821.
[86] M.F. Anjos, G. Yen, Provably near-optimal solutions for very large single-row facility layout problems, Optim. Methods Softw. 24 (2009) 805–817.