Journal Pre-proof

An efficient Variable Neighborhood Search for the Space-Free Multi-Row Facility Layout problem

Alberto Herrán, J. Manuel Colmenar, Abraham Duarte

PII: S0377-2217(21)00253-8
DOI: https://doi.org/10.1016/j.ejor.2021.03.027
Reference: EOR 17108

To appear in: *European Journal of Operational Research*

Received date: 13 November 2020
Accepted date: 13 March 2021

Please cite this article as: Alberto Herrán, J. Manuel Colmenar, Abraham Duarte, An efficient Variable Neighborhood Search for the Space-Free Multi-Row Facility Layout problem, *European Journal of Operational Research* (2021), doi: https://doi.org/10.1016/j.ejor.2021.03.027

Universidad Rey Juan Carlos

Escuela Superior de Ingeniería Informática

An efficient Variable Neighborhood Search for the Space-Free Multi-Row Facility Layout problem

Alberto Herrán, J. Manuel Colmenar, Abraham Duarte

Highlights

- Successful application of a metaheuristic to solve a challenging NP-Problem.
- Greedy randomized approach to construct the initial solutions.
- Efficient evaluation of move operators.
- Parallel implementation of the final algorithm.
- Thorough experimentation to tune the parameters and to evaluate the contribution of each algorithmic component.
- Comparison with the state-of-the-art methods over a set of 528 instances.

An efficient Variable Neighborhood Search for the Space-Free Multi-Row Facility Layout problem[☆]

Alberto Herrán, J. Manuel Colmenar, Abraham Duarte*

Dept. Computer Sciences, Universidad Rey Juan Carlos
C/. Tulipán, s/n, Móstoles, 28933 (Madrid), Spain

Abstract

The Space-Free Multi-Row Facility Layout problem (SF-MRFLP) seeks for a non-overlapping layout of departments (facilities) on a given number of rows satisfying the following constraints: no space is allowed between two adjacent facilities and the left-most department of the arrangement must have zero abscissa. The objective is to minimize the total communication cost among facilities. In this paper, a Variable Neighborhood Search (VNS) algorithm is proposed to solve this $\mathcal{NP}$-Hard problem. It has practical applications in the context of the arrangement of rooms in buildings, semiconductor wafer fabrication, or flexible manufacturing systems. A thorough set of preliminary experiments is conducted to evaluate the influence of the proposed strategies and to tune the corresponding search parameters. The best variant of our algorithm is tested over a large set of 528 instances previously used in the related literature. Experimental results show that the proposed algorithm improves the state-of-the-art methods, reaching all the optimal values or, alternatively, the best-known values (if the optimum is unknown) but in considerably shorter computing times. These results are further confirmed by conducting a Bayesian statistical analysis.

Keywords: Metaheuristics, Variable Neighborhood Search, Facility layout problem, Space-free Multi-row Facility Layout problem

[☆]This work has been partially supported by the Spanish Ministerio de Ciencia, Innovación y Universidades (MCIU/AEI/FEDER, UE) under grant ref. PGC2018-095322-B-C22; and Comunidad de Madrid y Fondos Estructurales de la Unión Europea with grant ref. P2018/TCS-4566.

*Corresponding author
Email address: alberto.herran@urjc.es, josemanuel.colmenar@urjc.es, abraham.duarte@urjc.es (Alberto Herrán, J. Manuel Colmenar, Abraham Duarte)

1. Introduction

Multi-Row Facility Location Problems (MRFLP) stands for a relevant family of combinatorial optimization problems where a set of departments (facilities) must be embedded within a determined number of rows in a feasible manner, i.e., each department must be completely located inside a row and not overlap with each other. In general, we have a pairwise non-negative weight for each pair of departments, being the aim of these problems the minimization of the total weighted center-to-center distances. Among all the different variants of MRFLP, this paper targets the Space-Free Multi-Row Facility Layout problem (SF-MRFLP) (Anjos & Vieira, 2017). As all the variants of the FLP family, it is a $\mathcal{NP}$-Hard problem (Loiola et al., 2007) where, in this case, the arrangement of departments on the rows shall start from a common point on the left end of the rows, and no space is allowed between two adjacent departments.

Departments may have different lengths but, without loss of generality, the height is the same for all of them and the distance among rows is considered negligible. Figure 1 shows an example where 13 departments are placed in 3 rows. For instance, departments `A`, `B`, `C`, and `D` are located in the first row. Similarly, departments `E`, `F`, `G`, and `H`, are located in the second row, and the rest of them in the third row. As it was aforementioned, the first department in each row (i.e., `A`, `E`, and `I`, respectively) is aligned to the left.

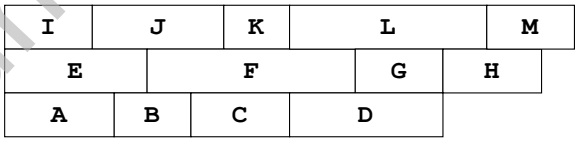

Figure 1: Example of layout with 13 departments placed in 3 rows.

The SF-MRFLP is a generalization of the Corridor Allocation Problem (CAP) which, under the same constraints of no space allowed between two adjacent facilities and aligning the left-most point of the arrangement of all rows, it only considers two rows (Amaral, 2012). Notice that these problems have been mainly approached in the literature separately. Amaral (2012) proposed exact and heuristic methods to deal with CAP. Specifically, a Mixed Integer Programming (MIP) formulation able to only

solve small-size instances (less than 15 facilities in several hours of execution time) and 3 heuristics based on 2-opt and 3-opt moves (solving instances with 50 facilities in a couple of hours). Ghosh & Kothari (2012) introduced two new hybrid procedures: a genetic algorithm coupled with a local search and a scatter search with path relinking. Experimental results showed that these algorithms solved instances from 25 to 50 facilities, where CPU time vary from 40 seconds to 4 hours. Hungerländer (2014) derived lower bounds for the CAP by using semidefinite programming (SDP). Additionally, the authors described a heuristic method to obtain feasible solutions by considering the SDP relaxation. The method provided tight global bounds in reasonable CPU time for instances with up to 15 facilities. Ahonen et al. (2014) presented a tabu search and a simulated annealing for the CAP. Reported results showed that the proposed methods reached the optimal (when available) or best-known solutions in previously used instances. The authors additionally introduced a set of larger and challenging instances. Finally, Fischer et al. (2019) proposed an exact method for the SF-MRFLP able to prove optimality in instances with up to 13 facilities and 5 rows.

The Double row layout problem (DRLP) is a particular case of the family of the multi-row layout problems where only two rows are considered. Notice that this problem is different to CAP since a feasible solution might have free space between consecutive facilities. Additionally, it is not required that the left walls of the leftmost facilities of each row are aligned. We refer the reader to Chae & Regan (2020) and Amaral (2020) for further details.

Both problems, CAP and SF-MRFLP, have relevant practical applications such as the arrangement of rooms in office buildings, hospitals, shopping centers or schools (see Amaral (2012) for further details). We can find other applications in the context of semiconductor wafer fabrication (Yang & Peters, 1997), flexible manufacturing systems (Bracht et al., 2017; Hassan, 1994), data memory layout (Wess & Zeitlhofer, 2004), or scheduling of parallel production lines (Geoffrion & Graves, 1976).

In this paper, we propose the application of an efficient metaheuristic implementation, based on Variable Neighborhood Search methodology, to the SF-MRFLP and its particular case, the CAP. To this aim, we have designed two constructive procedures and defined two neighborhood structures for the local search process. Besides, we propose a method to efficiently compute the objective function, which boosts the

exploration process. In addition, we have designed a parallel approach that allows our method to obtain solutions in very short computation times. Our approach is general enough to deal with instances with 2 or more rows ($m \geq 2$). Hence, we have compared our algorithm with the state-of-the-art methods of the CAP and the SF-MRFLP by considering a set of 132 previously studied instances in both problems. In all the comparisons, we have obtained equal or better results than the previous approaches in terms of quality, reaching all the known optimal values. Besides, our algorithm is able to produce the results in shorter computation times, obtaining in some cases up to 99% of time savings when considering a straightforward implementation. Additionally, we have also proposed a new benchmark with large and challenging instances with more than two rows to establish a new framework for future comparisons. Finally, we have performed a Bayesian analysis of performance between our proposal and the state-of-the-art methods which proves the superiority of our proposal.

The rest of the paper is organized as follows. Section 2 formally describes the optimization problem. Section 3 details the definition of the neighborhoods and their efficient exploration. Section 4 explains the algorithmic proposal, and sections 5 and 6 detail the experimental experience. Finally, Section 7 draws the conclusions and the future work on this problem.

2. Formal description of the problem

The SF-MRFLP can be formally described as an optimization problem that aims to minimize a given objective function $\mathcal{F}$. Hence, we describe in this section our proposal to represent a solution as well as the adaption of the objective function.

Given a set of D departments, a solution to this optimization problem can be described as a mapping φ that assigns the set of departments from D to the corresponding layout with m rows. Specifically, given a department $u \in D$, $\varphi(u) = (i, j)$ indicates that u is allocated in position j of row i. The mapping for the solution shown in Figure 1 will include $\varphi(\texttt{A}) = (1, 1)$, $\varphi(\texttt{B}) = (1, 2)$ and so on, ending with $\varphi(\texttt{M}) = (3, 5)$.

Let W be a $|D| \times |D|$ squared matrix, where w_{uv} is the weight between two departments $u, v \in D$ (which may indicate transportation, transmission, or communication costs) and let L_u be the length of department u. We define $\delta_{\varphi(u)}$ as the horizontal

distance from the left end of the layout φ to the center of the department u located at $j\text{-}th$ position in row i, as shown in Equation (1). Then, the objective function of SF-MRFLP, denoted with $\mathcal{F}$, can be computed as the addition of the relative distances among all departments multiplied by their pairwise weight, as shown in Equation (2). Notice that vertical distances do not contribute to the objective function.

$$\delta_{\varphi(u)} = \frac{L_u}{2} + \sum_{\substack{v \in D \\ \varphi(v)=(i,j') \\ 1 \leq j' < j}} L_v \quad where \quad \varphi(u) = (i,j) \tag{1}$$

$$\mathcal{F}(W,\varphi) = \sum_{\substack{u,v \in D \\ u \neq v}} w_{uv} \cdot \left| \delta_{\varphi(u)} - \delta_{\varphi(v)} \right| \tag{2}$$

The optimization problem then consists in finding the optimal solution $\varphi^\star$ that minimizes the aforementioned objective function. More formally,

$$\varphi^\star \leftarrow \arg\min_{\varphi \in \Phi} \mathcal{F}(W,\varphi) \tag{3}$$

where Φ is the set of all feasible department layouts.

3. Efficient exploration of the solution space

The intelligent exploration of the space of solutions performed by heuristics and metaheuristics play a key role in combinatorial optimization. This feature is crucial in problems such as the SF-MRFLP, which is $\mathcal{NP}$-Hard, and deals with a huge space of solutions. With the aim of performing an efficient exploration across candidate solutions of this problem, we have defined two neighborhood structures and an efficient computation of the objective function which contribute to drastically reduce the computation time.

3.1. Neighborhood structures

The representation of a solution in an optimization problem determines the way in which the neighborhoods of a given solution are defined. Specifically, neighborhoods contain the set of solutions that can be reached by performing a move over a particular solution. In this paper, we propose two different moves to produce the corresponding neighborhoods for the SF-MRFLP. The first one corresponds to the *exchange* between

two departments. Figure 2 shows an example where the positions of departments J and C are interchanged in the layout φ leading to a new solution φ' after the move. Interchanged departments are represented with black background, while departments that modify its previous location (with respect to the left end of the layout) are highlighted with light/dark grey background.

In general, given a solution φ and two departments $u, v \in D$, with $\varphi(u) = (i, j)$ and $\varphi(v) = (k, l)$, the move $exchange(\varphi, u, v)$ generates a new solution φ' where the positions of u and v have been exchanged: $\varphi'(u) = \varphi(v)$ and $\varphi'(v) = \varphi(u)$. In other words, the department u is removed from its current position in the layout (i, j) and inserted in (k, l). Simultaneously, the department v is removed from (k, l) and inserted in (i, j). For the sake of simplicity, we represent this move as $\varphi' \leftarrow exchange(\varphi, u, v)$.

The neighborhood generated with *exchange* moves, called $N_{exchange}$, is formally defined as shown in Equation (4).

$$N_{exchange}(\varphi) = \{ \quad \varphi' \leftarrow exchange(\varphi, u, v) : \\ \forall u, v \in D \ \wedge \ u \neq v \quad \} \tag{4}$$

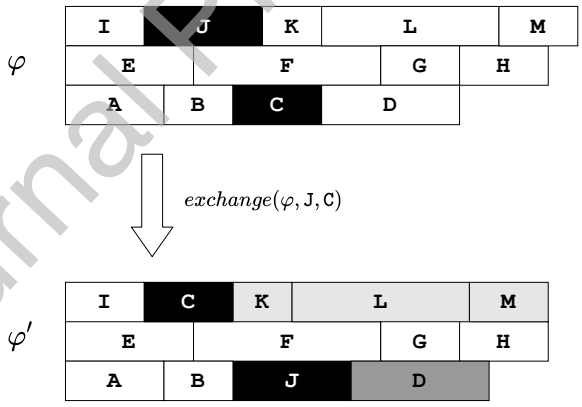

Figure 2: Effect of an *exchange* move.

The second proposed move is the *insertion* of a department in a different position of the layout. Figure 3 shows an example where the department J is inserted at position (1,3) of the layout φ leading to a new solution φ'. As in the previous example, the inserted department is highlighted with black background, while departments that modify their previous location (with respect to the left end of the layout) are

highlighted with light/dark grey background.

Therefore, given a solution φ and a department $u \in D$, with $\varphi(u) = (i, j)$, the move $insert(\varphi, u, k, l)$ removes the department u from its original position (i, j) and inserts it in a different position (k, l), reaching a new solution φ' where the department u is located at position (k, l), and departments located in positions higher than j in row i, and those located in positions higher than l in row k are shifted accordingly. For the sake of simplicity, we represent this move as $\varphi' \leftarrow insert(\varphi, u, k, l)$

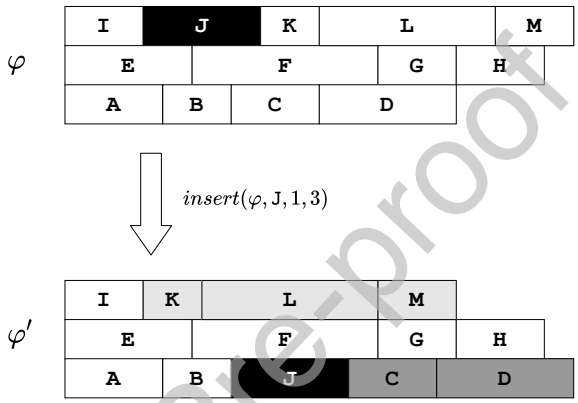

Figure 3: Effect of an *insert* move.

The neighborhood generated with *insert* moves, called N_{insert}, is formally defined as shown in Equation (5), where $|\varphi_k|$ is the number of departments in row k.

$$N_{insert}(\varphi) = \{ \quad \varphi' \leftarrow insert(\varphi, u, k, l) : \\ \forall u \in D \ \wedge \ 1 \leq k \leq m \ \wedge \ 1 \leq l \leq |\varphi_k| \quad \} \tag{5}$$

Notice that the size of $N_{exchange}$ and N_{insert} neighborhoods are $n(n-1)/2$ and $n(n+m-2)$ respectively, where $n = |D|$. Hence, if we take into account that the computational complexity of evaluating each solution from scratch is $O(n^2)$, a straightforward implementation of a scanning strategy for this neighborhood would result in an algorithm of computational complexity $O(n^4)$.

3.2. Efficient neighborhood exploration

One of the key aspects when designing an efficient move operator is how to compute the evaluation of neighbor solutions. In the case of SF-MRFLP, the objective function value of each neighbor solution $\varphi' \in N_{move}(\varphi)$, being *move* either *exchange*

or *insert*, can be efficiently obtained by just computing the cost variation for those departments u and v whose relative position has changed after the move is performed. To illustrate this idea, we have coloured different departments in Figures 2 and 3. In these examples, notice how the relative positions within departments in zones represented with the same color (white, light grey, dark grey or black) have not changed after the move. Hence, we only have to compute the cost change between departments with different background color, since the relative positions among them may have changed. In the previous figures, black departments are those involved in the move (`J` and `C` for the *exchange* move, and `J` for the *insert* move), white departments are those that remain at the same position after the move, and the grey ones are those whose positions may change after the move.

Therefore, the evaluation of a neighbor solution $\varphi' \in N_{move}(\varphi)$, after performing some move, can be efficiently computed as it is shown in Equation (6), where $\Delta_{move}(\varphi, \varphi')$ represents the cost change due to the move, which is only computed for departments u and v whose relative distance has change. That is, those located at different zones.

$$\mathcal{F}(W, \varphi') = \mathcal{F}(W, \varphi) + \Delta_{move}(\varphi, \varphi') \tag{6}$$

Considering the definition of Equation (1), $\delta_{\varphi(u)}$ (similarly, $\delta_{\varphi(v)}$) will be the horizontal distance from the left end of the layout to the center of department u (similarly, v). Let φ be a feasible layout of the SF-MRFLP and let $u, v \in D$ be two departments located in $\varphi(u)$ and $\varphi(v)$, respectively. Equation (7) defines the computation of $\Delta_{move}(\varphi, \varphi')$, considering the extension of the aforementioned δ-distance to φ'.

$$\Delta_{move}(\varphi, \varphi') = \sum_{\substack{u,v \in D \\ |\delta_{\varphi'(u)} - \delta_{\varphi'(v)}| \neq \\ |\delta_{\varphi(u)} - \delta_{\varphi(v)}|}} w_{uv} \cdot \left(\left| \delta_{\varphi'(u)} - \delta_{\varphi'(v)} \right| - \left| \delta_{\varphi(u)} - \delta_{\varphi(v)} \right| \right) \tag{7}$$

It is worth mentioning that this computation is different for both, *exchange* and *insert* moves. Additionally, it also depends on whether u and v are in the same row or not. The detail of the equations that will allow an efficient computation of the objective function in all these cases for both moves can be found in Appendix I (see Supplementary material).

4. Variable Neighborhood Search

The Variable Neighborhood Search (VNS) metaheuristic effectively combines the intensification provided by a local search method with the diversification implemented by a combination of a neighborhood change mechanism and a perturbation procedure (usually known as *shake*) which modifies the incumbent solution (Mladenović & Hansen, 1997). Among the different strategies within the VNS methodology (Hansen & Mladenović, 2014), we select in this work the Basic VNS (BVNS) variant since it is simple and provides really competitive results (Herrán et al., 2019a,b, 2020a,b).

4.1. Basic VNS

Algorithm 1 shows the pseudo-code for BVNS. In addition to the instance data D and W, the algorithm receives three additional parameters: α, β, and t_{max}. As seen, it starts with the generation of a solution φ by means of a constructive method (step 1). In this paper, the constructive method will be implemented by considering the Greedy Adaptive Search Procedure (GRASP) methodology (Feo & Resende, 1995; Feo et al., 1994), which is parametrized with the input parameter α (see Section 4.2 for further details). The largest explored neighborhood k_{max} is determined by the input parameter β. In order to make dependant this value to the size of the corresponding instance, it is computed as a percentage of the number of departments n (step 2). The input parameter t_{max} establishes the termination criterion. Instead of using the canonical definition of BVNS (i.e., time limit as in Hansen & Mladenović (2014)), our proposal consists of a loop guided by a maximum number of iterations (steps 3 to 12). As it is customary in BVNS designs, step 4 sets the initial value for k and the inner loop begins. Then, the incumbent solution is perturbed with the *shake* procedure (see Section 4.4) in step 6. After that, the resulting solution is improved in step 7 by considering any of the local search strategies described in Section 4.3.

Steps 8 to 12 correspond to the neighborhood change criterion. If the quality of the improved solution φ'' is better than the quality of φ, this solution is updated and k is reset. Otherwise, the size of the perturbation is increased to explore a larger neighborhood. Therefore, the returned solution φ will be the result of t_{max} iterations improving the solution returned by the constructive method.

Algorithm 1: BVNS($D, W, \alpha, \beta, t_{max}$)

```
1  φ ← Constructive(D, α)                                  // Section 4.2
2  k_max ← β · n
3  repeat t_max times
4      k ← 1
5      while k ≤ k_max do
6          φ' ← Shake(φ, k)                                // Section 4.4
7          φ'' ← LocalSearch(W, φ')                        // Section 4.3
8          if F(W, φ'') < F(W, φ) then
9              φ ← φ''
10             k ← 1
11         else
12             k ← k + 1
13 return φ
```

4.2. Constructive procedures

In this work, the constructive procedure builds an initial solution by considering a GRASP approach. It iteratively includes departments, one at a time, to a partial solution with less than n departments. We denote with φ_p as a partial solution of the SP-MRFLP where p departments have already been included in the layout under construction. Let P be the set that contains those mapped departments. Therefore, given a partial solution φ_p, the departments $u \in D \setminus P$ (i.e., those not mapped yet) are ranked according to the cost of the new partial solution that will be obtained after adding the department u to the layout. In particular, the new partial solution considers the best location for u in φ_p to minimize its partial cost $\mathcal{F}(W, \varphi_p)$. If the addition of a department to different rows has the same cost, it is finally inserted in the shortest row (ties are broken at random).

Algorithm 2 shows the pseudo-code of the GRASP constructive method. A candidate list CL is created in step 1 including all departments of the instance. Then, in order to initialize the solution, a department is randomly selected and placed in the

first position of an empty partial solution, φ_p, as seen in steps 2 and 3. The selected department is removed from CL in step 4 and the main loop begins, iterating till all the departments are located. The next department to be included in φ_p is selected in step 6. However, two different approaches are considered: *Greedy-Random* and *Random-Greedy*, which will be next described. Once the department is selected, it is placed in the best position of φ_p according to the objective function value (the lower the better, since SF-MRFLP is a minimization problem), and removed from CL, as seen in steps 7 and 8, respectively. Finally, the solution is returned in step 9.

Algorithm 2: GRASP constructive algorithm (D, α)

1 $CL \leftarrow D$
2 $u \leftarrow$ `SelectRandom`(CL)
3 $\varphi_p(1,1) \leftarrow u$
4 $CL \leftarrow CL \setminus \{u\}$
5 **while** $|CL| > 0$ **do**
6 $\quad u \leftarrow \begin{cases} \texttt{Greedy-Random}(CL, \alpha) \\ \texttt{Random-Greedy}(CL, \alpha) \end{cases}$
7 $\quad \varphi_p \leftarrow$ `IncludeBestLocation`(φ_p, u)
8 $\quad CL \leftarrow CL \setminus \{u\}$
9 **return** φ_p

The two methods to select the department to be added to the current solution, *Greedy-Random* and *Random-Greedy,* produce two GRASP constructive methods called $C1$ and $C2$, respectively. Both strategies use the same greedy function to select the department u to be included in the new partial solution, $g(u) = \mathcal{F}(W, \varphi_p)$, where φ_p is the partial solution after inserting u in the best possible location in the layout.

Algorithm 3 shows the pseudo-code of $C1$, which is the *Greedy-Random* approach. As seen, steps 2 and 3 obtain the minimum and maximum values of the greedy function for all the departments in the candidate list. Then, a restricted candidate list RCL is created including those departments whose greedy function value is below a threshold determined by α. Notice that α balances the greediness-randomness of the selection,

since $\alpha = 0$ corresponds to a fully random selection and $\alpha = 1$ corresponds to a fully greedy selection. Finally, a department u is randomly selected from RCL and returned to the constructive method, as seen in steps 5 and 6.

Algorithm 3: *Greedy-Random* selection.

1 **function** $C1(CL, \alpha)$**:**
2 $\quad g_{min} = \min_{u \in CL} g(u)$
3 $\quad g_{max} = \max_{u \in CL} g(u)$
4 $\quad RCL \leftarrow \{u \in CL : g(u) \leq g_{max} - \alpha \cdot (g_{max} - g_{min})\}$
5 $\quad u \leftarrow$ `SelectRandom`(RCL)
6 $\quad$ **return** u

Similarly, Algorithm 4 shows the pseudo-code of the method used to select the candidate department with a *Random-Greedy* strategy, namely $C2$. As in $C1$, $\alpha = 0$ corresponds to a fully random selection, while $\alpha = 1$ corresponds to a fully greedy selection, since α determines the number of solutions randomly selected to form the RCL (steps 2 and 3). Step 4 returns the department with the minimum value of the greedy function among the selected ones, which is finally returned in step 5.

Algorithm 4: *Random-Greedy* selection.

1 **function** $C2(CL, \alpha)$**:**
2 $\quad size \leftarrow$ `max`$(\lfloor \alpha \cdot |CL| \rfloor, 1)$
3 $\quad RCL \leftarrow$ `SelectRandomSet`$(CL, size)$
4 $\quad u \leftarrow \arg\min_{u \in RCL} g(u)$
5 $\quad$ **return** u

In addition to these two constructive methods, a straightforward strategy that constructs a feasible solution at random is also analyzed. This procedure is extremely fast and, additionally, it might introduce diversity in the set of constructed solutions. On the other hand, it usually provides low quality solutions when considering the objective function value. Specifically, the method selects at random one department at a time and then, it adds the department to the partial solution at a random

location. We use this method, denoted as $C0$, as a baseline to compare our more elaborated constructive methods.

4.3. Local search strategies

There exist two typical strategies to explore a neighborhood: *best* improvement and *first* improvement. The former explores all of the solutions in the neighborhood by a fully deterministic procedure, and the best move (i.e., the one that leads to a solution φ' with minimum associated cost) is applied at each iteration. The *first* improvement strategy explores the neighborhood of an initial solution and performs the first move that enhances the resulting cost. The procedure usually chooses different moves at random in order to obtain diverse solutions, and halts when the entire neighborhood has been explored and no moves are able to improve the cost of the initial solution.

In addition to this two typical approaches, in this work an *hybrid* local search method is also applied (Herrán et al., 2020a). This method combines the *first* and *best* strategies. Given a particular solution, the idea consists of selecting a department at random in each step (similarly to the *first* improvement strategy), and then finding the best move only for such department (as carried out by the *best* improvement strategy). If the value of the objective function does not decrease, the algorithm randomly chooses a different department and looks for the new best move for that department. This process is initialized and repeated as it finds better solutions. Finally, the algorithm reaches a local minimum and halts when no move operations can improve the cost of a previous solution.

In order to describe the neighborhood exploration in a general way, the pseudo-code of the proposed local search method is shown in Algorithm 5. It receives as input parameters the instance data W and a feasible constructed solution φ. This method systematically explores the neighborhoods of φ generated by the specified move (*exchange* or *insert*) according to the selected strategy (*Best*, *Hybrid*, or *First*) until no improvement is meet.

The implementation details of `ExploreNeighborhood` method depend on the selected move and exploration strategy. In this paper, we consider a composed exploration of the two aforementioned neighborhoods. Specifically, we determine for $N_{exchange}$ the best scanning strategy (*Best*, *Hybrid*, or *First*). We proceed similarly

Algorithm 5: Local Search (W, φ)

```
improve ← true
while improve do
    improve ← false
    φ' ← ExploreNeighborhood(φ)
    if F(W, φ') < F(W, φ) then
        improve ← true
        φ ← φ'
return φ
```

with N_{insert}. Then, `ExploreNeighborhood` actually executes the best move between these two neigborhoods.

4.4. Shake procedures

We propose different implementations of the *shake* procedure based on the *exchange* and *insert* moves. In particular, six different shake procedures have been implemented, depending on the type of moves used for this purpose. Given the current value of the perturbation, k, the first procedure, S_{1exc}, executes k consecutive *exchange* moves at random, returning the corresponding perturbed solution.

The second proposal, called S_{2exc}, behaves similarly, but only performs *horizontal exchanges*. That is, S_{2exc} selects, at random, a department $u \in D$ located in row i and column j. Then, this method executes the exchange between u and another department v also located in row i and any column $l \neq j$. This process is repeated k times to have the corresponding perturbed solution.

The third method, S_{3exc}, tries to perform *vertical exchanges*. Hence, it randomly selects a department $u \in D$, with $\varphi(u) = (i, j)$. Then, it randomly selects a department v in a different row (i.e., $r \neq i$) that is located in a column "overlapped" by u. For example, consider again Figure 1. Department `J` located in the second position of the third row, $\varphi(\texttt{J}) = (3, 2)$ could be exchanged with departments `A`, `B`, `C`, `E` and `F`. Similarly, department `M` can only be exchanged with `H`. As in previous shake methods, this random move is executed k times to produced the perturbed solution.

In addition to these three methods, the counterparts S_{1ins}, S_{2ins}, and S_{3ins} are also proposed, behaving similarly to S_{1exc}, S_{2exc} and S_{3exc}, but using *insert* moves instead. For the sake of brevity, we omit the description.

5. Preliminary experiments

Our experimental experience is developed on 132 instances taken from Ghosh & Kothari (2012); Ahonen et al. (2014); Fischer et al. (2019) with different values of m from 2 to 5, giving a total number of 528 instances. A full description of these instances can be found in Appendix II (see Supplementary material). Moreover, in order to facilitate future comparisons, we have made the full set publicly available at http://grafo.etsii.urjc.es/.

In this work, we propose the application of the VNS methodology to the SF-MRFLP. Specifically, we propose two constructive methods, two neighborhood structures explored with three different strategies, an efficient evaluation of the objective function, and six shake procedures. We additionally propose several parallel schemes for our final BVNS proposal (Herrán et al., 2020b). In order to determine the best combination of strategies and parameter values for the final algorithm, we have conducted a set of preliminary experiments. To this aim we have considered a representative subset of 15 test instances from the whole set of them, with different sizes and properties, in order to avoid the over-fitting effect. These instances are highlighted in bold in Tables 1, 2 and 3 in Appendix II (see Supplementary material). Moreover, we split each instance into 4 different instances with different values of m from 2 to 5, having a total of 60 test instances (i.e., 12%) from the whole set of 528 instances. The remaining set of instances is reserved for the final comparison with the state-of-the-art procedures.

All our methods were coded in C++ and the experiments were run on a laptop computer provided with an Intel Core-i7 6500U processor running at 2.5 GHz with 8 Gb of RAM using Windows 10. The code will be available on the website of our research group: https://grafo.etsii.urjc.es/. Next, we will describe the set of experiments we have conducted in this paper.

5.1. Constructive methods comparison

The aim of the first experiment is to select the best constructive procedure for the SF-MRFLP. Hence, we have compared the results of the two proposed constructive methods described in Section 4.2 under different values for the input parameter α. In particular, we have tested $\alpha = \{0.25, 0.5, 0.75\}$, and the generation of a new random value for α on each construction, denoted as *rnd*. We also include a totally random procedure, denoted as $C0$, which produces feasible solutions. The idea to include this basic procedure is to evaluate the contribution of including more elaborated strategies.

Table 1 shows the average cost for 30 executions of each constructive algorithm, the average number of times the algorithm reached the best value in this experiment for a given instance (#Best), and the average execution time in seconds (CPU(s)). As we can see, all the configurations of the proposed methods $C1$ and $C2$ clearly outperform the purely random method $C0$. In particular, $C1(0.75)$ seems to be the best one among the all greedy variants.

Table 1: Constructive methods comparison.

Method	α	**Cost**	**#Best**	**CPU(s)**
$C0$	-	203939.0	0	**0.0008**
$C1$	0.25	136060.8	5	0.0249
	0.50	135672.6	10	0.0248
	0.75	**135288.6**	**19**	0.0247
	rnd	135758.6	2	0.0258
$C2$	0.25	136253.8	7	0.0066
	0.50	136114.2	4	0.0126
	0.75	135811.7	9	0.0181
	rnd	136072.8	4	0.0109

Once $C0$ is discarded, in order to select the best greedy constructive procedure among all the variants of $C1$ and $C2$, we have coupled all of them with a local search which explores both, the *exchange* and *insert* neighborhoods, starting from the best solution found in 30 independent constructions. Specifically, it returns the best solution found between these two independent neighborhood explorations. Moreover, this local search follows a *Best* strategy to fully explore these two neighborhoods. We denote this local search as LS. Table 2 shows the averaged results for 30 executions of each greedy constructive variant coupled with the local search. As seen in the results, the final cost of the solutions is very similar among the studied configurations.

However, the best one in terms of cost and number of best solutions reached is $C2$ with α=0.5. Hence, this configuration will be selected for the following experiments.

Table 2: Greedy constructive procedures coupled with a local search.

Method	α	Cost	#Best	CPU(s)
$C1 + LS$	0.25	126276.9	7	0.3511
	0.50	126293.9	7	0.3451
	0.75	126274.2	6	0.3409
	rnd	126242.4	7	0.3490
C2 + LS	0.25	126224.3	10	**0.3356**
	0.50	**126213.3**	**11**	0.3382
	0.75	126271.1	7	0.3377
	rnd	126239.6	5	0.3379

5.2. Efficient computation of moves

In addition to the performance analysis of the different algorithmic configurations, the efficient computation of the neighborhoods generated using *exchange* and *insert* moves separately was also analyzed (see Section 3.2 and Appendix I in the Supplementary material). To this purpose we compare the CPU time of both, *Straightforward* and *Efficient* implementations of the local search procedures based on the *exchange* and *insert* moves, starting from the same initial solution constructed with $C2(0.5)$. Table 3 shows the averaged results for 30 executions of each experiment. As we can see, in both cases the efficient computation of moves reaches the same solution in less computation time than the straightforward evaluation of solutions.

Notice that, with a straightforward evaluation of moves, *exchanges* are usually faster than *insertions*, since the the size of the N_{insert} neighborhood is twice the size of $N_{exchange}$. However, with the efficient computation, the local search based on *insert* moves is faster than the counterpart based on *exchange* moves. The efficient computation of *insert* moves reaches a 77.1% of improvement by using the strategy depicted in Figure 5 of Appendix I in the Supplementary material (i.e., exploring all the possible insertions by performing consecutive swap moves), while the exploration based on *exchange* reaches a 25.4% of time savings. Therefore, the *Efficient* computation of moves is proven to be much faster that the *Straightforward* computation, and will be used in the experiments from now on.

Table 3: Efficient computation of cost for both, *exchange* and *insert*, neighborhoods.

Neighborhood	Cost	$CPU_{Straight.}$(s)	$CPU_{Efficient}$(s)	Improv.(%)
$N_{Exchange}$	126481.9	0.123	0.085	25.4%
N_{Insert}	128614.8	0.169	0.024	77.1%

5.3. Neighborhood exploration strategies

In this experiment, we study the performance of different exploration strategies used at the local search stage: best improvement (*Best*), first improvement (*First*), and the hybrid approach (*Hybrid*). To this purpose, we have executed a multi-start version of each local search implementation using the constructive method $C2(0.5)$ to produce the initial solution of each iteration. Instead of running all the algorithms for a fixed number of restarts, we have first executed the slowest strategy for 100 restarts for each neighborhood, *Best* for *exchange* moves and *First* for *insert* moves, obtaining a time limit for the other strategies in each case (*First* and *Hybrid* for *exchange*, and *Best* and *Hybrid* for *insert* moves). Table 4 shows the results for both neighborhoods in the same table. As we can see, the *First* strategy achieves the best results for *exchange* moves, while the *Hybrid* strategy is the best performer in the case of *insert* moves.

Table 4: Neighborhood exploration strategies comparison.

Move	Strategy	Cost	#Best	Iter.	CPU(s)
Exchange	*Best*	123717.4	8	100	8.42
	First	**123494.1**	**45**	**266**	8.42
	Hybrid	123532.0	13	261	8.42
Insert	*Best*	124766.1	24	201	4.83
	First	125073.7	2	100	4.83
	Hybrid	**124766.0**	**35**	**300**	4.83

According to these results, the *First* strategy will be used for the exploration of the *exchange* moves, while the *Hybrid* one will be applied for *insert* moves in the VNS proposal.

5.4. BVNS performance

In this experiment we analyze the performance of different *Shake* methods for the BVNS algorithm, as explained in Section 4.4. To this purpose we start with a BVNS schema running a local search using the best neighborhood exploration strategies

determined in the previous experiments for both, *exchange* and *insert* moves. Hence, at each iteration, BVNS explores the *exchange* and *insert* neighborhoods following the *First* and *Hybrid* strategies respectively, and both starting from the same incumbent solution. Then, it returns the best solution found among these two independent neighborhood explorations.

In order to decide the best *Shake* procedure, we run six different BVNS configurations according to the six different *Shake* procedures explained in Section 4.4. In this case, we state the parameter $\beta = 0.4$ for all variants and $t_{max} = 10$ outer iterations as stopping condition. Table 5 shows the results of this experiment. As it can be seen, the best performance is achieved by perturbing the incumbent solution with horizontal moves in both moves: S_{2exc} and S_{2ins}. Regarding the CPU times, all the variants present similar values.

Table 5: Performance of BVNS for different *Shake* procedures with $\beta = 0.4$ and $t_{max} = 10$.

Shake	**Cost**	**#Best**	**CPU(s)**
S_{1exc}	123259.6	17	5.89
$\boldsymbol{S_{2exc}}$	**123247.5**	**41**	5.87
S_{3exc}	123811.5	4	**5.73**
S_{1ins}	123306.3	25	5.77
$\boldsymbol{S_{2ins}}$	**123302.4**	**30**	5.63
S_{3ins}	123419.7	7	**5.43**

Once the best procedures have been identified, another experiment is run in order to explore lower values for the β parameter when running the same CPU time as in the case of $\beta = 0.4$ for the two best performers. Table 6 shows the results of this experiment for $\beta = \{0.1, 0.2, 0.3, 0.4\}$. As we can see, reducing the value of β improves the BVNS performance, being S_{2exc} with $\beta = 0.1$ the best variant among all.

At this point, the constructive, local search and shake phases of the BVNS approach have been tuned to reach the best performance.

5.5. Parallel VNS performance

This section is devoted to fix the final setting of our best BVNS algorithm identified in the previous section. Firstly, a termination criteria is defined based on a number of iterations. Next, we explore if a multi-start version of the algorithm is better than a single long run. And, finally, we parallelize the multi-start BVNS in order to get the

Table 6: Performance of BVNS variants with S_{2exc} and S_{2ins} for different values of β from 0.1 to 0.4.

Shake	β	Cost	#Best	CPU(s)
$\boldsymbol{S_{2exc}}$	**0.1**	**123179.4**	**22**	5.87
	0.2	123180.4	14	5.87
	0.3	123212.1	10	5.87
	0.4	123247.5	7	5.87
S_{2ins}	0.1	123286.0	6	5.63
	0.2	123287.9	5	5.63
	0.3	123294.2	3	5.63
	0.4	123306.3	2	5.63

best possible performance when it is executed in a computer with several processors. Moreover, we study different communication strategies among the different algorithms when they run in parallel.

In order to test the influence of the number of iterations, t_{max}, over the whole performance of the algorithm, we depict in Figure 4 the average cost obtained on the test instances at each iteration from 1 to $t_{max} = 2000$. As we can see, the algorithm convergence stabilizes as it approaches to 2000 iterations, being the improvement after this value marginal. Therefore, we use this value of $t_{max} = 2000$ to establish the stopping criterion of our algorithm.

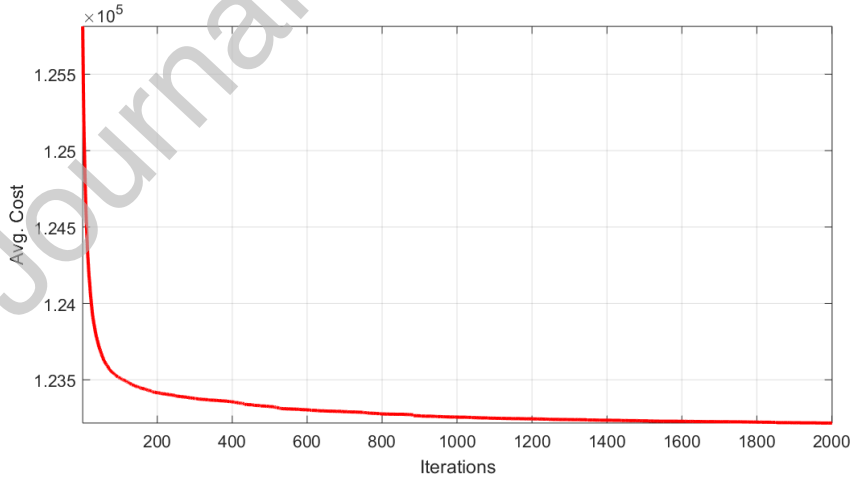

Figure 4: Convergence of BVNS with the number of iterations.

In the next experiment, we study if a multi-start strategy (i.e. multiple short

runs of a BVNS) performs better than a single long run of a single BVNS with a total number of iterations $t_{max} = 2000$. We call this multi-start version of the previous algorithm MS-BVNS. Specifically, we compare a single BVNS with $t_{max} = 2000$ against a MS-BVNS restarting the search from a new constructed solution after t_{max}/R iterations. In this case, several values of the number of restarts R are studied, from 5 to 40. Notice that all MS-BVNS variants are executed the same total number of iterations as the single BVNS version. Table 7 shows that, among the studied values, the best result is obtained with $R = 20$.

Table 7: Performance of different MS-BVNS algorithms for different values of R (from 5 to 40).

R	Cost	#Best	CPU(s)
1	123156.4	11	**30.69**
5	123133.6	13	31.15
10	123124.9	20	31.48
20	**123124.2**	**39**	31.89
30	123143.5	15	32.35
40	123152.7	8	33.81

Finally, we further analyze a third alternative, called Parallel BVNS (PBVNS), which consists in taking advantage of the parallel architecture of current processors. In particular, this new variant uses 4 processors to simultaneously execute the $R = 20$ independent runs in parallel. Therefore, each processor will execute 5 BVNS algorithms by means of different threads. Moreover, this approach allows us to implement a last schema based on the migration of the best solutions among different processors every K iterations. This strategy allows the algorithm to intensify the search around promising regions. Table 8 shows the results of an experiment where we test the most common communication topologies (*Fully-Connected*, *Ring*, and *Master-Slave*) for several values of K. Besides, these configurations are compared also with the parallel version of the best MS-BVNS variant in the previous experiment (without communication among processors).

In light of the results, the *Master-Slave* topology with $K = 10$ offers the best performance over all the tested algorithm configuration.

Table 8: Performance of different PBVNS variants.

Communication	K	Cost	#Best	CPU(s)
None	-	123124.2	9	19.01
Fully	5	123111.5	6	**18.80**
	10	123105.94	16	19.03
	20	123105.76	13	18.96
Ring	5	123107.9	13	18.87
	10	123106.1	15	19.01
	20	123111.1	10	18.99
Master-Slave	5	123107.2	15	19.03
	10	**123105.5**	**20**	19.04
	20	123109.9	11	19.11

6. Experimental results

In this section, we compare our best VNS variant identified after the thorough preliminary experimentation, PBVNS with $C2(0.5)$, a local search based on a *First* and *Hybrid* efficient explorations of *exchange* and *insert* moves respectively, S_{2exc} with $\beta = 0.1$ and *Master-Slave* communication topology with $K = 10$, with the current state-of-the-art methods. Moreover, our PBVNS executes $R = 20$ independent runs in parallel with $t_{max} = 2000$ as termination criterion.

6.1. Comparison with state-of-the-art methods for $m = 2$

Firstly, we compare our approach with the current state-of-the-art methods for $m = 2$, i.e. SF-DRFLP or CAP. Specifically, we compare our PBVNS with the exact and heuristic approaches in Amaral (2012), the different population heuristics described in Ghosh & Kothari (2012), the Tabu Search and Simulated Annealing approaches described in Ahonen et al. (2014), and the permutation-based Genetic Algorithm (pGA) proposed in Kalita et al. (2019).

Table 9 shows a comparison among our PBVNS proposal with the exact and heuristic algorithms proposed in Amaral (2012), denoted as *Amaral 2012*, and the Genetic Algorithm and Scatter Search methods described in Ghosh & Kothari (2012), denoted as CAP-GA and CAP-SS, for several instances with sizes from $n = 9$ to $n = 15$ reported in Ghosh & Kothari (2012) (see Table 1 of Appendix II in the Supplementary material). In the second column, the exact algorithm proposed in Amaral (2012) provides the optimal solution, denoted with a $\star$, for instances with

$9 \leq n \leq 13$. In the case of instance `Am15`, the heuristic proposed in Amaral (2012) gives an upper bound to the cost of an optimal solution which was better than the output by CPLEX 12.1.0 after 8.6 hours of execution. The following columns of the table present the cost of the best solutions achieved by the GA and SS heuristics reported in Ghosh & Kothari (2012), and those obtained by our PBVNS proposal in one run. We also report the execution time in seconds for each instance and algorithm. As we can see in this table, our proposal is able to reach the optimal values or the same upper bounds as the previous approaches for all the instances in this set, spending less execution time. In the case of the state-of-the-art methods reported in Ghosh & Kothari (2012), they conducted the experimentation on a Intel Core-i5 2400 processor at 3.1 GHz which, according to `http://www.cpubenchmark.net/`, obtains a benchmark result which is 16% better than our i7 processor. Hence, our PBVNS also has competitive CPU times compared to the fastest previous approaches for this set. Notice that the computing time should be only considered in the comparison of metaheuristic algorithms (i.e., PBVNS, CAP-GA and CAP-SS). The exact method may spend a significant amount of time on proving the optimality.

Table 9: Comparison with exact and heuristic algorithms in Amaral (2012) and population heuristics in Ghosh & Kothari (2012) for instances with sizes $9 \leq n \leq 15$.

	Amaral 2012		**CAP-GA**		**CAP-SS**		**PBVNS**	
Instance	**Cost**	**CPU(s)**	**Cost**	**CPU(s)**	**Cost**	**CPU(s)**	**Cost**	**CPU(s)**
S9	**1181.5**⋆	18.42	**1181.5**	17.61	**1181.5**	0.19	**1181.5**	**0.07**
S9H	**2294.5**⋆	1145.94	**2294.5**	17.54	**2294.5**	0.24	**2294.5**	**0.07**
S10	**1374.5**⋆	62.75	**1374.5**	21.96	**1374.5**	0.27	**1374.5**	**0.08**
S11	**3439.5**⋆	496.22	**3439.5**	29.21	**3439.5**	0.53	**3439.5**	**0.10**
Am12a	**1529.0**⋆	1869.31	**1529.0**	40.20	**1529.0**	0.90	**1529.0**	**0.12**
Am12b	**1609.5**⋆	1412.06	**1609.5**	38.30	**1609.5**	0.80	**1609.5**	**0.11**
Am13a	**2467.5**⋆	10298.01	**2467.5**	50.30	**2467.5**	1.50	**2467.5**	**0.12**
Am13b	**2870.0**⋆	5144.13	**2870.0**	53.76	**2870.0**	1.35	**2870.0**	**0.12**
Am15	**3195.0**	2.41	**3195.0**	77.99	**3195.0**	2.18	**3195.0**	**0.14**
Avg.	**2217.9**	2272.14	**2217.9**	38.54	**2217.9**	0.88	**2217.9**	**0.10**

We also report in Table 10 our computational experience with PBVNS against GA and SS algorithms for a set of larger 36 benchmark instances with sizes $30 \leq n \leq 49$ reported in Ghosh & Kothari (2012) (see Table 1 of Appendix II in the Supplementary material). In this case, we have executed our PBVNS 10 times reporting the best cost found after these 10 executions, together with the total CPU time consumed by these 10 executions of each instance. As we can see in this table, PBVNS obtains the best

values for all the instances in this set, while CAP-GA only obtains the best value in 14 (out of 36) instances and CAP-SS in 19. Regarding the CPU time, despite the fact that our computer has a lower performance, the average execution time of PBVNS is 99.09% faster than CAP-GA and 94.76% faster than CAP-SS.

Table 10: Comparison with population heuristics in Ghosh & Kothari (2012) for instances with sizes $30 \leq n \leq 49$.

	CAP-GA		CAP-SS		PBVNS	
Instance	**Cost**	**CPU(s)**	**Cost**	**CPU(s)**	**Cost**	**CPU(s)**
N25_01	**2302.0**	**618.58**	**2302.0**	**41.50**	**2302.0**	**21.10**
N25_02	**18595.5**	**736.25**	**18595.5**	**59.91**	**18595.5**	**15.29**
N25_03	**12114.0**	**677.39**	**12114.0**	**63.42**	**12114.0**	**15.25**
N25_04	**24192.5**	**742.12**	**24192.5**	**56.20**	**24192.5**	**15.60**
N25_05	**7819.0**	**719.97**	**7819.0**	**53.14**	**7819.0**	**14.28**
N30_01	**4115.0**	1506.69	**4115.0**	118.68	**4115.0**	**19.69**
N30_02	**10779.5**	1565.07	**10779.5**	150.06	**10779.5**	**19.40**
N30_03	**22702.0**	1611.19	**22702.0**	163.10	**22702.0**	**21.18**
N30_04	**28401.5**	1628.00	**28401.5**	189.20	**28401.5**	**22.57**
N30_05	**57400.0**	1672.68	**57400.0**	188.97	**57400.0**	**22.73**
P1	**30282.5**	2287.84	**30282.5**	285.32	**30282.5**	**28.73**
P2	**33974.0**	2349.55	33978.0	221.66	**33974.0**	**31.55**
P3	35065.5	2330.13	35060.5	322.39	**35052.5**	**31.21**
P4	34671.5	2967.47	34673.5	478.29	**34666.5**	**35.72**
P5	30803.0	2801.70	30781.0	390.76	**30771.0**	**32.98**
P6	34425.5	3010.98	**34424.5**	446.38	**34424.5**	**33.77**
ste36_01	**4966.0**	3475.61	**4966.0**	423.74	**4966.0**	**28.05**
ste36_02	88262.0	3237.71	87489.0	721.35	**87390.0**	**43.29**
ste36_03	50696.5	3279.40	50127.5	674.71	**50084.5**	**39.70**
ste36_04	46890.5	3446.11	46824.5	504.18	**46737.5**	**41.91**
ste36_05	44783.5	3407.52	**44488.5**	593.57	**44488.5**	**39.55**
N40_01	53763.5	5597.52	**53722.5**	782.26	**53722.5**	**51.92**
N40_02	48916.0	5448.85	**48908.0**	773.86	**48908.0**	**55.56**
N40_03	39259.5	5330.11	39255.5	760.12	**39250.5**	**54.45**
N40_04	38355.0	5459.45	**38354.0**	741.42	**38354.0**	**53.70**
N40_05	51523.0	5585.37	51507.0	836.69	**51496.0**	**54.20**
sko42_01	**12731.0**	7174.64	**12731.0**	1249.04	**12731.0**	**39.20**
sko42_02	108049.5	6815.77	108020.5	1533.29	**108018.5**	**72.78**
sko42_03	86667.5	6854.25	86667.5	956.28	**86644.5**	**67.30**
sko42_04	68769.0	6509.35	68733.0	1141.34	**68701.0**	**65.55**
sko42_05	124099.5	7052.59	124058.5	1236.47	**124017.5**	**70.48**
sko49_01	20472.0	14934.38	20479.0	2683.93	**20470.0**	**51.07**
sko49_02	208270.0	14338.86	208081.0	2733.87	**208059.0**	**119.07**
sko49_03	162267.0	14351.91	162196.0	2877.19	**162193.0**	**116.21**
sko49_04	118311.5	14290.03	118264.5	3716.16	**118246.5**	**111.16**
sko49_05	332990.0	14913.18	**332855.0**	2596.39	**332855.0**	**122.30**
Avg.	65569.8	5652.71	65494.4	983.57	**65480.7**	**51.52**

Finally, we perform a larger comparison with the Tabu Search (TS) and Simulated Annealing (SA) algorithms reported in Ahonen et al. (2014), and the permutation-based Genetic Algorithm (pGA) proposed in Kalita et al. (2019) over the same sets of instances with $m = 2$ used in this work (see Table 2 of Appendix II in the Sup-

plementary material, where the work Kalita et al. (2019) only provides cost values for the *Extra* set). Tables 11 to 14 show the detailed results of our PVNS approach for the *Small*, *Medium*, *Large* and *Extra* sets of instances from Ahonen et al. (2014). For each instance, each algorithm is executed 30 times, reporting average values for both, cost and CPU time. In these tables, best results are highlighted with bold font. Given that columns 2 to 7 average the results, we have also included in the last two columns of the tables the cost value of the best solution found after the 30 executions among all the algorithms, together with the algorithm that reaches that value. If there is more than one algorithm providing the best solution, we use bold font for the fastest algorithm.

Table 11: Comparison with TS and SA from Ahonen et al. (2014) for *Small* instances with $m = 2$.

	Average Cost			Average CPU (s)			Best Cost	
Name	**TS**	**SA**	**PBVNS**	**TS**	**SA**	**PBVNS**	**Value**	**Algorithm**
S9	**1181.5**	**1181.5**	**1181.5**	1.14	4.05	**0.09**	1181.5	TS, SA, **PBVNS**
S9H	**2294.5**	**2294.5**	**2294.5**	1.15	3.46	**0.10**	2294.5	TS, SA, **PBVNS**
S10	**1374.5**	**1374.5**	**1374.5**	1.58	4.88	**0.11**	1374.5	TS, SA, **PBVNS**
S11	**3439.5**	**3439.5**	**3439.5**	2.02	5.97	**0.13**	3439.5	TS, SA, **PBVNS**
Am12a	**1529.0**	**1529.0**	**1529.0**	2.43	7.36	**0.15**	1529.0	TS, SA, **PBVNS**
Am12b	**1609.5**	**1609.5**	**1609.5**	2.43	7.28	**0.14**	1609.5	TS, SA, **PBVNS**
Am13a	**2467.5**	**2467.5**	**2467.5**	2.99	8.55	**0.16**	2467.5	TS, SA, **PBVNS**
Am13b	**2870.0**	**2870.0**	**2870.0**	3.01	8.74	**0.15**	2870.0	TS, SA, **PBVNS**
Am15	**3195.0**	**3195.0**	**3195.0**	4.70	12.02	**0.19**	3195.0	TS, SA, **PBVNS**
Avg.	**2217.9**	**2217.9**	**2217.9**	2.38	6.92	**0.14**	2217.9	

Table 12: Comparison with TS and SA from Ahonen et al. (2014) for *Medium* instances with $m = 2$.

	Average Cost			Average CPU (s)			Best Cost	
Name	**TS**	**SA**	**PBVNS**	**TS**	**SA**	**PBVNS**	**Value**	**Algorithm**
N30_01	**4115.0**	**4115.0**	**4115.0**	26.51	4.41	**2.60**	4115.0	TS, SA, **PBVNS**
N30_02	10780.4	10788.8	**10779.6**	37.68	4.40	**2.54**	10779.5	TS, SA, **PBVNS**
N30_03	22703.7	22712.8	**22702.1**	35.33	4.45	**2.79**	22702.0	TS, SA, **PBVNS**
N30_04	28403.4	28416.7	**28401.5**	36.35	4.66	**2.97**	28401.5	TS, SA, **PBVNS**
N30_05	57415.2	57420.5	**57414.2**	38.67	4.58	**3.01**	57400.0	TS, SA, **PBVNS**
Avg.	24683.5	24690.8	**24682.5**	34.91	4.50	**2.78**	24679.6	

Table 15 summarizes the results of the previous tables (11 to 14), reporting the average cost and execution time of each algorithm, and the number of times each algorithm obtained the best result (#Best). As it can be seen, our PVNS approach obtains the best results in cost for all instances in sets *Small* and *Medium*, spending less execution time. Specifically, for these sets of instances, the average execution times for PBVNS is 0.14 seconds for set *Small* and 2.78 seconds and for set *Medium*,

Table 13: Comparison with TS and SA from Ahonen et al. (2014) for *Large* instances with $m = 2$.

	Average Cost			Average CPU (s)			Best Cost	
Name	**TS**	**SA**	**PBVNS**	**TS**	**SA**	**PBVNS**	**Value**	**Algorithm**
sko42_01	12733.6	12733.8	**12731.0**	111.49	25.87	**3.90**	12731.0	TS, SA, **PBVNS**
sko42_02	108065.9	108048.5	**108022.8**	111.26	26.40	**7.24**	108006.5	PBVNS
sko42_03	86681.2	86672.2	**86651.1**	122.54	22.90	**6.71**	86644.5	PBVNS
sko42_04	68747.4	68770.5	**68710.6**	134.90	23.00	**6.54**	68701.0	PBVNS
sko42_05	124081.6	124069.4	**124043.1**	123.43	23.28	**7.05**	124017.5	SA, **PBVNS**
sko49_01	20479.9	20474.2	**20471.5**	151.20	48.17	**5.10**	20470.0	TS, SA, **PBVNS**
sko49_02	208260.9	208146.6	**208100.0**	151.20	48.53	**11.90**	208056.0	PBVNS
sko49_03	162337.1	162258.5	**162212.3**	139.72	48.96	**11.60**	162182.0	PBVNS
sko49_04	118340.9	118288.9	**118267.0**	136.40	49.29	**11.14**	118246.5	PBVNS
sko49_05	333087.3	332909.5	**332874.6**	142.48	48.64	**12.16**	332836.0	PBVNS
sko56_01	31978.5	31973.0	**31972.0**	235.88	93.36	**7.44**	31972.0	TS, SA, **PBVNS**
sko56_02	248391.4	248243.6	**248230.9**	230.74	93.94	**19.46**	248219.0	PBVNS
sko56_03	85293.0	85223.7	**85193.4**	240.03	94.91	**16.59**	85184.0	PBVNS
sko56_04	156835.6	156715.7	**156677.6**	209.38	138.41	**18.51**	156646.0	PBVNS
sko56_05	296424.8	296247.3	**296220.4**	225.92	151.11	**19.80**	296176.5	SA
AKV_n_60_05	159838.8	159721.6	**159685.4**	281.22	310.01	**28.24**	159643.0	PBVNS
sko64_05	251048.9	250901.1	**250885.7**	351.54	418.83	**32.06**	250870.5	PBVNS
AKV_n_70_05	2111623.3	2110035.0	**2109926.2**	459.60	657.10	**58.65**	2109745.5	PBVNS
Avg.	254680.6	254524.1	**254493.1**	197.72	129.04	**15.78**	254463.8	

which corresponds to a reduction of a 94.3% and 38.2% in relation to the fastest previous approaches in these sets. Regarding the *Large* set, PBVNS obtains the best results in 17 (out 18) instances, with a CPU time improvement of 89.5% and 83.9% over TS and SA respectively. Finally, PBVNS gets the best result in 31 (out of 36) instances in the *Extra* set with a reduction of a 88.8% and 75.1% in CPU time over TS and SA respectively. In the case of the state-of-the-art methods reported in Ahonen et al. (2014), they conducted the experimentation on an Intel Core2 Quad P9550 processor running at 2.83 GHz and 4 Gb of RAM. According to `http://www.cpubenchmark.net/`, our Core-i7 processor obtains a benchmark result which is 31.32% better than Core2 Quad. However, the comparison of the running time of different algorithms is always a tough task since it depends on the programming language, computer, programmer skills, etc. Then, we only analyze this metric from a qualitative perspective.

6.2. Comparison with state-of-the-art methods for $m > 2$

Next, we test our algorithm on instances with $m > 2$, i.e. SF-MRFLP. For this purpose, we compare our PBVNS proposal with the exact algorithm described in Fischer et al. (2019) over the same set of instances reported in this work, with sizes $5 \leq n \leq 13$, where the optimal values are known (see Table 3 of Appendix II in the

Table 14: Results for *Extra* instances in Ahonen et al. (2014) with $m = 2$.

	Average Cost				Average CPU (s)			Best Cost	
Name	**TS**	**SA**	**pGA**	**PBVNS**	**TS**	**SA**	**PBVNS**	**Value**	**Algorithm**
CAP_n60_p30_s30_1	204442.2	204174.3	204173.5	**204142.9**	281.70	136.08	**27.22**	204089.0	PBVNS
CAP_n60_p30_s30_2	193586.5	193324.0	193310.9	**193229.8**	314.67	135.63	**26.32**	193199.5	PBVNS
CAP_n60_p30_s30_3	161876.0	161546.7	161533.3	**161526.8**	353.98	137.58	**26.38**	161475.5	pGA, PBVNS
CAP_n60_p30_s40_1	135438.8	135213.2	**135171.3**	135175.2	345.10	137.88	**27.42**	135133.5	SA, pGA
CAP_n60_p30_s40_2	159390.2	159256.1	159230.0	**159189.8**	344.07	135.90	**26.69**	159114.0	SA, pGA
CAP_n60_p30_s40_3	159223.5	159174.7	**158973.5**	159015.1	365.87	137.47	**26.83**	158967.5	PBVNS
CAP_n60_p30_s50_1	110755.1	110609.1	**110514.1**	110536.6	345.28	138.17	**26.94**	110493.5	pGA, PBVNS
CAP_n60_p30_s50_2	115539.4	115382.9	**115320.3**	115347.1	312.46	136.87	**25.96**	115302.0	PBVNS
CAP_n60_p30_s50_3	114505.0	114342.5	114287.4	**114188.0**	291.27	139.33	**27.44**	114137.0	PBVNS
CAP_n60_p30_s60_1	108243.5	108154.2	108139.4	**108137.9**	274.27	136.07	**25.40**	108117.0	PBVNS
CAP_n60_p30_s60_2	110142.5	110036.7	110040.4	**109948.7**	319.85	138.00	**25.11**	109908.5	PBVNS
CAP_n60_p30_s60_3	92023.8	91937.4	**91652.2**	91759.3	350.38	137.12	**24.85**	91619.0	pGA, PBVNS
CAP_n60_p60_s30_1	446018.3	445604.3	**445380.4**	445399.9	300.92	135.37	**26.87**	445377.5	pGA, PBVNS
CAP_n60_p60_s30_2	408511.2	408006.4	407962.9	**407907.5**	305.08	133.87	**26.55**	407890.5	PBVNS
CAP_n60_p60_s30_3	417188.0	416994.6	**416933.8**	416940.8	258.02	135.13	**26.15**	416912.5	PBVNS
CAP_n60_p60_s40_1	313715.1	313383.1	313383.0	**313330.6**	291.15	134.80	**25.71**	313304.0	PBVNS
CAP_n60_p60_s40_2	320924.8	320794.6	320802.7	**320772.7**	276.00	136.92	**26.11**	320758.5	SA, pGA, PBVNS
CAP_n60_p60_s40_3	363410.1	363121.7	363185.3	**363082.1**	326.67	134.27	**25.44**	363020.5	SA, pGA, PBVNS
CAP_n60_p60_s50_1	273649.7	273478.5	**273422.8**	273457.8	297.62	136.61	**25.55**	273420.0	pGA, PBVNS
CAP_n60_p60_s50_2	270009.9	269865.4	**269682.8**	269723.5	305.78	137.11	**25.58**	269680.5	pGA, PBVNS
CAP_n60_p60_s50_3	295664.0	295484.1	**295423.9**	295428.2	271.78	134.56	**24.85**	295419.0	PBVNS
CAP_n60_p60_s60_1	228070.6	227968.9	**227897.8**	227909.5	287.13	137.59	**24.49**	227896.0	pGA, PBVNS
CAP_n60_p60_s60_2	246847.5	246581.9	246570.6	**246557.3**	329.51	135.66	**24.38**	246523.0	SA, pGA
CAP_n60_p60_s60_3	206747.5	206576.6	**206496.2**	206513.0	319.77	133.46	**24.31**	206486.5	PBVNS
CAP_n60_p90_s30_1	629193.8	628970.1	629012.4	**628883.4**	275.36	134.75	**25.86**	628850.0	PBVNS
CAP_n60_p90_s30_2	561556.1	561529.6	561663.8	**561193.2**	289.42	133.86	**26.46**	561170.5	SA, pGA, PBVNS
CAP_n60_p90_s30_3	587971.7	587846.5	**587827.6**	587831.7	256.79	134.37	**26.73**	587816.5	PBVNS
CAP_n60_p90_s40_1	474203.6	474073.8	474100.4	**474055.3**	255.20	135.65	**25.49**	474046.0	SA, pGA, PBVNS
CAP_n60_p90_s40_2	480232.4	480162.9	480241.7	**479970.4**	282.45	135.32	**25.78**	479963.0	SA, pGA, PBVNS
CAP_n60_p90_s40_3	512555.6	512470.6	512495.9	**512459.5**	270.03	135.13	**26.09**	512452.0	pGA, PBVNS
CAP_n60_p90_s50_1	479869.1	479815.6	479771.2	**479705.3**	307.11	135.36	**24.89**	479683.0	PBVNS
CAP_n60_p90_s50_2	445393.5	445182.0	445212.5	**445121.7**	281.36	133.36	**24.89**	445030.0	pGA, PBVNS
CAP_n60_p90_s50_3	495407.9	495341.9	495130.7	**495113.2**	296.83	133.48	**24.99**	495052.5	PBVNS
CAP_n60_p90_s60_1	385570.0	385455.5	385458.8	**385435.4**	273.86	134.30	**23.81**	385430.5	SA, pGA, PBVNS
CAP_n60_p90_s60_2	344895.3	344801.8	344811.4	**344779.1**	334.60	137.16	**24.46**	344775.0	SA, pGA, PBVNS
CAP_n60_p90_s60_3	411429.2	411296.7	411269.3	**411213.3**	256.57	133.93	**23.95**	411205.0	pGA, PBVNS
Avg.	312894.5	312721.0	312680.1	**312638.4**	301.33	135.78	**25.72**		

Table 15: Aggregated comparison with TS and SA from Ahonen et al. (2014).

Instances	**Algorithm**	**Avg. Cost**	**Avg. CPU (s)**	**#Best**
Small (9)	TS	**2217.9**	2.38	**9**
	SA	**2217.9**	6.92	**9**
	PBVNS	**2217.9**	**0.14**	**9**
Medium (5)	TS	24683.5	34.91	**5**
	SA	24690.8	4.50	**5**
	PBVNS	**24682.5**	**2.78**	**5**
Large (18)	TS	254680.6	197.72	3
	SA	254524.1	129.04	5
	PBVNS	**254493.1**	**15.78**	**17**
Extra (36)	TS	312894.5	301.33	0
	SA	312721.0	135.78	10
	pGA	312680.1	-	20
	PBVNS	**312638.4**	**25.72**	**31**

Supplementary material). Results are shown in Table 16 where, for each instance, we report the CPU time (in seconds) of each algorithm to reach the optimal solution, shown in column Cost. Our algorithm obtains the optimal cost value for all the instances in less CPU time. Specifically, we obtain a reduction over 99% for all the tested m values. Moreover, all experiments in Fischer et al. (2019) were performed on a Quad-Core Intel Core-i7 4770 running at 3.4 GHz with 32 GB RAM in single processor mode, and using CPLEX 12.8 as an IP solver, which, according to `http://www.cpubenchmark.net/`, obtains a benchmark result which is 215% better than our i7 processor.

6.3. New results for large instances with $m > 2$

Finally, given the reduced computation time of our algorithm, we provide in this section new results for $m > 2$ on larger instances with the aim to extend the current state-of-the-art for the SF-MRFLP. For this purpose, we use the same instance sets *Medium*, *Large* and *Extra* described in Ahonen et al. (2014), but with values of m from 3 to 5. After executing 30 runs on each instance, we report the best cost value, average cost value and average CPU time in seconds for each instance in tables 17 to 19.

6.4. Statistical analysis

In order to statistically analyze the results of our proposal in relation to the previous approaches, we have conducted a performance analysis comparison of multiple algorithms over multiple instances simultaneously under the Bayesian approach described in Calvo et al. (2018, 2019). This kind of analysis is able to produce a ranking of algorithms based on a probability distribution created after the analysis of the results. Therefore, it calculates the expected probability of each algorithm to obtain better results than the others, denoted as probability of winning, giving credible intervals to this probability.

Given that we performed two comparisons with the state-of-the-art methods, next we describe these analysis separately.

Figure 5 shows the credible intervals (5% and 95% quantiles) and the expected probability of winning for the proposed PBVNS algorithm and the CAP-GA and

Table 16: Results for instances in Fischer et al. (2019) with $m > 2$.

	m=3			m=4			m=5		
Name	**$CPU_{Fischer}$(s)**	**CPU_{PVNS}(s)**	**Cost**	**$CPU_{Fischer}$(s)**	**CPU_{PVNS}(s)**	**Cost**	**$CPU_{Fischer}$(s)**	**CPU_{PVNS}(s)**	**Cost**
Am11a	221.85	**0.91**	3763.5	497.47	**0.77**	2761.5	663.31	**0.62**	2156.5
Am11b	157.30	**0.84**	2403.5	462.33	**0.83**	1816.5	720.08	**0.71**	1446.5
Am11c	192.02	**0.79**	2504.5	512.12	**0.72**	1906.5	687.41	**0.52**	1438.5
Am11d	126.52	**0.75**	532.5	336.00	**0.81**	389.5	565.26	**0.72**	305.5
Am11e	188.00	**0.93**	416.0	368.72	**0.92**	298.0	613.11	**0.92**	254.0
Am11f	184.26	**0.78**	558.0	413.51	**0.73**	408.0	568.83	**0.67**	297.0
Am12a	648.28	**0.78**	1028.0	2648.94	**0.87**	785.0	4341.10	**0.70**	629.0
Am12b	798.60	**0.78**	1117.5	2678.24	**0.88**	855.5	4707.03	**0.69**	617.5
Am12c	522.02	**0.75**	1344.0	2362.38	**0.69**	1009.0	4173.54	**0.67**	806.0
Am12d	425.86	**0.77**	734.0	1567.27	**0.71**	530.0	3039.42	**0.61**	417.0
Am12e	462.27	**0.75**	701.0	1533.48	**0.81**	505.0	2875.37	**0.73**	391.0
Am12f	548.66	**0.78**	661.0	1726.60	**0.64**	452.0	3396.61	**0.65**	366.0
Am13a	2058.19	**0.94**	1569.5	8909.18	**0.76**	1145.5	19346.42	**0.64**	882.5
Am13b	2631.50	**0.77**	1904.0	10129.68	**0.77**	1372.0	21744.91	**0.66**	1085.0
Am13c	2074.15	**0.83**	2659.0	9738.88	**0.71**	1933.0	20597.51	**0.51**	1456.0
Am13d	2013.40	**0.75**	4069.5	11617.26	**0.77**	3086.5	22380.68	**0.83**	2361.5
Am13e	2605.13	**0.86**	4340.5	13190.14	**0.83**	3233.5	26320.47	**0.68**	2502.5
Am13f	2296.95	**0.76**	5011.5	15025.67	**0.67**	3847.5	29431.88	**0.78**	3095.5
HA5	0.02	**0.01**	34.5	0.02	**0.01**	34.5	0.02	**0.01**	34.5
HA6	0.16	**0.07**	110.5	0.20	**0.08**	98.5	0.20	**0.07**	94.5
HA7	0.38	**0.13**	108.0	0.50	**0.49**	77.0	0.50	**0.39**	61.0
HA8	2.24	**0.65**	138.0	2.82	**0.41**	98.0	3.51	**0.51**	82.0
HA9	10.46	**0.59**	317.5	13.05	**0.61**	212.5	17.74	**0.61**	157.5
HA10	49.50	**0.75**	541.0	72.10	**0.55**	363.0	101.90	**0.56**	272.0
HA11	157.98	**0.77**	547.0	360.90	**0.70**	394.0	522.87	**0.63**	294.0
HA12	832.92	**0.74**	674.0	2083.55	**0.69**	483.0	3482.10	**0.65**	376.0
HA13	2970.84	**0.79**	1019.5	12500.24	**0.70**	750.5	25311.54	**0.68**	611.5
Small.09-1	19.27	**0.55**	1532.0	26.82	**0.52**	1152.0	23.41	**0.47**	870.0
Small.09-2	14.91	**0.46**	1398.0	21.90	**0.52**	1052.0	22.29	**0.51**	823.0
Small.09-3	17.81	**0.49**	1389.0	25.57	**0.51**	1039.0	22.17	**0.52**	779.0
Small.09-4	19.00	**0.55**	1393.5	22.30	**0.49**	1017.5	22.65	**0.51**	781.5
Small.09-5	30.68	**0.57**	2457.5	27.35	**0.45**	1866.5	24.64	**0.49**	1399.5
Small.09-6	34.46	**0.54**	2553.0	28.61	**0.46**	1914.0	25.76	**0.44**	1441.0
Small.09-7	22.65	**0.52**	2529.0	28.93	**0.54**	1905.0	23.42	**0.51**	1404.0
Small.09-8	24.93	**0.58**	2587.5	28.80	**0.44**	1917.5	24.82	**0.48**	1457.5
Small.09-9	9.40	**0.60**	579.0	21.41	**0.45**	450.0	17.99	**0.48**	318.0
Small.09-10	18.39	**0.55**	691.5	18.84	**0.42**	485.5	20.47	**0.57**	362.5
Small.10-1	32.10	**0.52**	918.5	66.51	**0.49**	653.5	97.72	**0.56**	534.5
Small.10-2	35.51	**0.58**	968.5	66.01	**0.48**	666.5	102.63	**0.47**	560.5
Small.10-3	26.32	**0.63**	933.0	64.82	**0.45**	667.0	101.96	**0.48**	556.0
Small.10-4	23.97	**0.60**	834.0	59.69	**0.43**	596.0	94.67	**0.43**	497.0
Small.10-5	28.42	**0.56**	458.5	65.72	**0.54**	349.5	95.21	**0.58**	279.5
Small.10-6	34.00	**0.64**	519.5	69.32	**0.54**	374.5	100.38	**0.50**	307.5
Small.10-7	31.20	**0.70**	382.5	63.71	**0.53**	297.5	99.18	**0.53**	237.5
Small.10-8	21.36	**0.60**	328.0	48.38	**0.53**	245.0	80.83	**0.48**	197.0
Small.10-9	39.72	**0.64**	619.5	84.96	**0.56**	447.5	105.17	**0.55**	351.5
Small.10-10	35.00	**0.69**	547.5	83.28	**0.54**	413.5	107.90	**0.54**	331.5
S9	8.34	**0.46**	771.5	18.05	**0.50**	587.5	21.98	**0.46**	472.5
S9H	19.64	**0.48**	1431.5	26.57	**0.44**	1062.5	22.12	**0.54**	817.5
S10	27.89	**0.54**	887.5	64.03	**0.50**	630.5	98.07	**0.47**	499.5
S11	167.88	**0.82**	2308.5	523.70	**0.68**	1743.5	687.37	**0.56**	1350.5
Avg.	449.46	**0.64**	1329.9	1966.79	**0.59**	987.8	3875.61	**0.56**	767.0

Table 17: Results for *Medium* instances in Ahonen et al. (2014) with $m > 2$.

	m=3			m=4			m=5		
Name	**Best Cost**	**Avg. Cost**	**Avg. CPU(s)**	**Best Cost**	**Avg. Cost**	**Avg. CPU(s)**	**Best Cost**	**Avg. Cost**	**Avg. CPU(s)**
N30_01	2721.0	2721.0	1.51	2043.0	2043.0	1.25	1616.0	1616.0	1.15
N30_02	7162.5	7165.4	1.80	5346.5	5347.3	1.69	4280.5	4280.5	1.64
N30_03	15100.0	15101.8	1.99	11327.0	11332.8	1.99	9052.0	9052.0	1.88
N30_04	18904.5	18905.5	2.18	14274.5	14276.5	2.13	11399.5	11399.8	2.05
N30_05	38326.0	38335.3	2.20	28323.0	28358.0	2.15	22917.0	22949.0	2.11

Table 18: Results for *Large* instances in Ahonen et al. (2014) with $m > 2$.

	m=3			m=4			m=5		
Name	**Best Cost**	**Avg. Cost**	**Avg. CPU(s)**	**Best Cost**	**Avg. Cost**	**Avg. CPU(s)**	**Best Cost**	**Avg. Cost**	**Avg. CPU(s)**
sko42_01	8458.0	8458.0	3.52	6353.0	6353.7	3.30	5072.0	5072.1	3.18
sko42_02	71984.5	71996.1	7.63	53991.5	54029.0	7.38	43111.5	43136.4	7.14
sko42_03	57826.5	57851.9	7.08	43385.5	43399.6	6.98	34539.5	34563.3	6.76
sko42_04	45746.0	45759.1	6.72	34241.0	34267.8	6.46	27334.0	27373.9	6.32
sko42_05	82436.5	82438.7	7.48	61795.5	61804.6	7.39	49312.5	49339.6	7.21
sko49_01	13625.0	13625.6	5.09	10196.0	10196.1	5.09	8132.0	8133.1	5.06
sko49_02	138604.0	138665.5	12.31	103722.0	103777.3	12.24	82928.0	82955.4	11.83
sko49_03	108088.0	108109.6	12.00	80958.0	80995.6	11.99	64720.0	64760.4	11.51
sko49_04	78889.5	78923.9	11.56	58947.5	58985.8	11.54	47209.5	47244.8	10.91
sko49_05	221507.0	221628.9	12.67	166192.0	166256.9	12.74	132610.0	132688.0	12.46
sko56_01	21319.0	21320.0	7.86	15922.0	15922.0	8.17	12753.0	12753.0	8.09
sko56_02	165190.0	165245.1	19.62	123827.0	123859.8	19.66	99083.0	99135.9	19.42
sko56_03	56790.0	56817.1	16.84	42596.0	42627.7	16.91	34089.0	34110.0	16.41
sko56_04	104386.0	104430.4	19.05	78268.0	78300.6	19.45	62557.0	62581.8	18.97
sko56_05	197488.5	197535.9	20.89	148194.5	148242.1	21.15	118213.5	118307.9	20.54
AKV_n_60_05	106615.0	106643.7	29.90	80168.0	80227.8	30.29	64338.0	64389.5	28.94
sko64_05	167100.5	167130.3	31.49	125399.5	125444.5	31.59	100225.5	100252.0	31.43
AKV_n_70_05	1407802.5	1408319.1	54.27	1057214.5	1057745.9	55.66	846244.5	847018.7	55.88

Table 19: Results for *Extra* instances in Ahonen et al. (2014) with $m > 2$.

	m=3			m=4			m=5		
Name	**Best Cost**	**Avg. Cost**	**Avg. CPU(s)**	**Best Cost**	**Avg. Cost**	**Avg. CPU(s)**	**Best Cost**	**Avg. Cost**	**Avg. CPU(s)**
CAP_n60_p30_s30_1	136259.0	136354.3	27.45	102256.0	102339.6	27.90	81598.0	81678.6	26.72
CAP_n60_p30_s30_2	128819.5	128888.3	26.94	96671.5	96742.3	26.98	77306.5	77387.9	26.86
CAP_n60_p30_s30_3	107756.5	107842.0	27.12	80691.5	80734.1	27.56	64635.5	64723.5	26.46
CAP_n60_p30_s40_1	90081.5	90169.0	27.48	67426.5	67521.5	26.80	54024.5	54109.9	25.90
CAP_n60_p30_s40_2	106088.0	106176.5	26.59	79572.0	79680.9	26.73	63768.0	63846.8	26.18
CAP_n60_p30_s40_3	105983.5	106151.8	26.52	79534.5	79607.2	26.76	63846.5	63928.7	25.69
CAP_n60_p30_s50_1	73619.5	73776.5	26.71	55413.5	55692.4	26.10	44500.5	44681.3	25.05
CAP_n60_p30_s50_2	76987.0	77063.9	25.76	58118.0	58188.7	25.99	46754.0	46791.9	25.47
CAP_n60_p30_s50_3	76050.0	76151.5	27.06	57191.0	57269.3	26.35	45768.0	45819.6	25.50
CAP_n60_p30_s60_1	72104.0	72134.6	25.25	53958.0	54015.7	24.27	43165.0	43221.7	23.41
CAP_n60_p30_s60_2	73262.5	73317.4	24.97	54889.5	54972.3	25.35	43914.5	43936.6	24.65
CAP_n60_p30_s60_3	61143.0	61208.6	24.61	45775.0	45860.6	23.80	36565.0	36656.8	22.83
CAP_n60_p60_s30_1	297045.5	297093.2	27.08	222830.5	222891.8	27.69	178399.5	178461.7	27.28
CAP_n60_p60_s30_2	271926.5	271983.9	26.26	203955.5	204040.1	26.93	163134.5	163174.7	25.91
CAP_n60_p60_s30_3	277988.5	278037.3	27.31	208403.5	208479.6	28.82	166676.5	166739.3	27.38
CAP_n60_p60_s40_1	209327.0	209355.8	26.24	157185.0	157235.8	26.48	125990.0	126023.7	25.78
CAP_n60_p60_s40_2	213817.5	213874.1	26.55	160716.5	160769.9	26.48	129162.5	129231.9	26.58
CAP_n60_p60_s40_3	242451.5	242520.6	26.38	182069.5	182130.2	27.59	145696.5	145817.9	27.70
CAP_n60_p60_s50_1	182457.0	182501.2	26.26	137129.0	137184.4	26.18	110323.0	110354.9	25.77
CAP_n60_p60_s50_2	180057.5	180176.6	26.15	135418.5	135563.9	26.42	108724.5	108920.8	26.26
CAP_n60_p60_s50_3	197012.0	197079.3	25.06	147860.0	147889.7	25.05	118525.0	118575.3	25.19
CAP_n60_p60_s60_1	151802.0	151851.1	24.42	113837.0	113866.3	24.22	91058.0	91096.6	23.80
CAP_n60_p60_s60_2	164228.0	164290.8	24.82	123198.0	123264.1	24.27	98397.0	98449.1	24.25
CAP_n60_p60_s60_3	137672.5	137732.7	24.27	103283.5	103342.8	23.82	82640.5	82721.5	23.30
CAP_n60_p90_s30_1	419051.0	419098.1	26.70	314451.0	314504.5	27.22	251594.0	251671.7	26.99
CAP_n60_p90_s30_2	374116.5	374145.7	27.39	281067.5	281124.0	27.71	225094.5	225133.6	27.16
CAP_n60_p90_s30_3	392105.5	392134.1	27.71	294254.5	294286.4	27.76	235770.5	235795.4	28.01
CAP_n60_p90_s40_1	316473.0	316517.6	26.77	237072.0	237082.1	27.15	190179.0	190198.6	27.51
CAP_n60_p90_s40_2	319977.0	320006.8	26.90	240433.0	240523.8	27.41	192624.0	192665.1	26.92
CAP_n60_p90_s40_3	341951.0	341989.5	27.08	256637.0	256654.3	28.17	206020.0	206044.6	28.55
CAP_n60_p90_s50_1	320161.0	320211.6	25.64	240309.0	240335.4	26.60	192971.0	193029.9	26.45
CAP_n60_p90_s50_2	297184.0	297247.1	26.36	223673.0	223874.2	26.68	179952.0	180026.7	26.54
CAP_n60_p90_s50_3	330279.5	330292.7	25.85	248443.5	248490.7	26.21	199824.5	199866.9	26.09
CAP_n60_p90_s60_1	256986.5	257008.7	24.08	192646.5	192658.3	24.19	154042.5	154064.7	23.73
CAP_n60_p90_s60_2	229964.0	229972.6	25.15	172508.0	172523.4	25.86	137819.0	137880.5	25.74
CAP_n60_p90_s60_3	273884.0	273917.3	23.65	205255.0	205304.7	23.81	164016.0	164038.5	23.82

CAP-SS ones. As it can be seen in the figure, PBVNS reaches the highest probability of obtaining the best solution, 0.587, in relation to the state-of-the-art methods. Moreover, the overall performance of the algorithms is clearly distinguished, because the credible intervals are not overlapped.

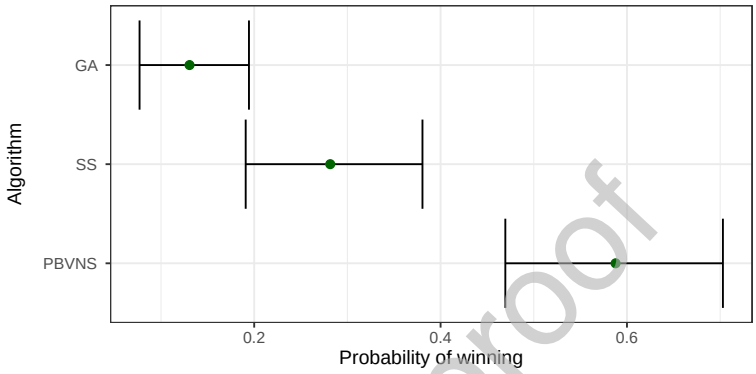

Figure 5: Probability of winning among the algorithms from Ghosh & Kothari (2012) and PBVNS.

Figure 6 shows the credible intervals (5% and 95% quantiles) and the expected probability of winning for the proposed PBVNS algorithm as well as the TS, SA , and pGA ones. As it can be seen in the figure, the credible intervals are not overlapped. Hence, the overall performance of the algorithms is clearly distinguished, being our PBVNS the algorithm with the highest probability of obtaining the best solution, 0.651, in relation to the state-of-the-art methods.

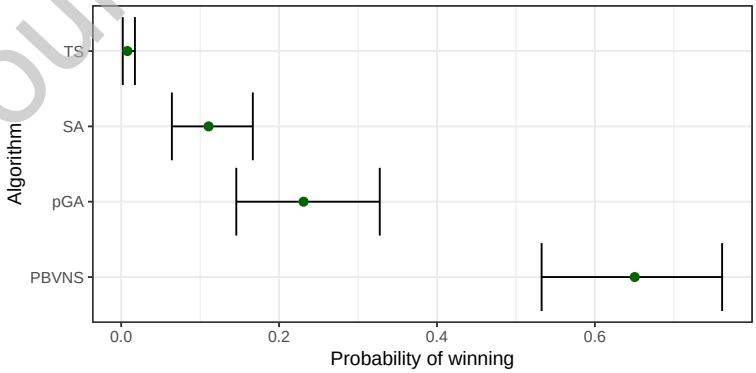

Figure 6: Probability of winning among the algorithms from Ahonen et al. (2014) , Kalita et al. (2019) and PBVNS.

Hence, the superiority of our method is not only supported by the results, but for this Bayesian analysis.

7. Conclusions and future work

In this work we have studied the application of the Variable Neighborhood Search methodology to solve the Space-Free Multiple Row Facility Location Problem SF-MRFLP and its particularization to two rows, known as the Corridor Allocation Problem CAP. To this aim, we have proposed two constructive procedures, three local search strategies and six shake methods that have been combined under a Basic VNS approach. Besides, we have designed an effective computation of the cost function that boosts the performance of the neighborhood exploration. Finally, we have proposed the parallelization of the algorithm, studying three different strategies.

The final configuration of the algorithm has been reached after a thorough preliminary experimentation on a set of test instances. The resulting configuration has been tested over a large set of 528 instances taken from the different works that formed the state of the art for CAP and SF-MRFLP. In the experimental experience we prove that our proposal is able to improve the state of the art results, reaching also all the known optimal values, but considerably reducing the execution time. We have also proven the superiority of our proposal be means of a Bayesian statistical analysis.

In order to facilitate future comparison with new metaheuristic proposals, we have introduced a set of 108 large and challenging instances, with 60 facilities and number of rows ranging from 3 to 5.

Currently, we are working on the adaption of this approach to other related location problems such as the Multiple Row Equal Facility Layout Problem. Additionally, we consider that there exits an avenue for future research in Multi-Floor Layout Problems as well as in Double Row Layout Problems from a metaheuristic perspective.

Acknowledgements

This work has been partially supported by the Spanish Ministerio de Ciencia, Innovación y Universidades (MCIU/AEI/FEDER, UE) under grant ref. PGC2018-

095322-B-C22; and Comunidad de Madrid y Fondos Estructurales de la Unión Europea with grant ref. P2018/TCS-4566.

References

Ahonen, H., de Alvarenga, A., & Amaral, A. (2014). Simulated annealing and tabu search approaches for the corridor allocation problem. *European Journal of Operational Research*, *232*, 221 – 233.

Amaral, A. (2020). A mixed-integer programming formulation of the double row layout problem based on a linear extension of a partial order. *Optimization Letters*, (pp. 1–17).

Amaral, A. R. (2012). The corridor allocation problem. *Computers and Operations Research*, *39*, 3325 – 3330.

Anjos, M. F., & Vieira, M. V. (2017). Mathematical optimization approaches for facility layout problems: The state-of-the-art and future research directions. *European Journal of Operational Research*, *261*, 1 – 16.

Bracht, U., Dahlbeck, M., Fischer, A., & Krüger, T. (2017). Combining simulation and optimization for extended double row facility layout problems in factory planning. In *Simulation Science* (pp. 39–59). Springer.

Calvo, B., Ceberio, J., & Lozano, J. A. (2018). Bayesian inference for algorithm ranking analysis. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* GECCO '18 (pp. 324–325). New York, NY, USA: ACM.

Calvo, B., Shir, O. M., Ceberio, J., Doerr, C., Wang, H., Bäck, T., & Lozano, J. A. (2019). Bayesian performance analysis for black-box optimization benchmarking. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* GECCO '19 (pp. 1789–1797). New York, NY, USA: ACM.

Chae, J., & Regan, A. C. (2020). A mixed integer programming model for a double row layout problem. *Computers Industrial Engineering*, *140*, 106244.

Feo, T., & Resende, M. (1995). Greedy randomized adaptive search procedures. *Journal of Global Optimization*, *6*, 109–133.

Feo, T., Resende, M., & Smith, S. (1994). A greedy randomized adaptive search procedure for maximum independent set. *Operations Research*, *42*, 860–878.

Fischer, A., Fischer, F., & Hungerlnder, P. (2019). New exact approaches to row layout problems. *Math. Program. Comput.*, *11*, 703–754.

Geoffrion, A. M., & Graves, G. W. (1976). Scheduling parallel production lines with changeover costs: Practical application of a quadratic assignment/lp approach. *Operations Research*, *24*, 595–610.

Ghosh, D., & Kothari, R. (2012). *Population Heuristics for the Corridor Allocation Problem*. IIMA Working Papers WP2012-09-02 Indian Institute of Management Ahmedabad, Research and Publication Department.

Hansen, P., & Mladenović, N. (2014). Variable neighborhood search. In *Search methodologies* (pp. 313–337). Springer.

Hassan, M. M. (1994). Machine layout problem in modern manufacturing facilities. *The International Journal of Production Research*, *32*, 2559–2584.

Herrán, A., Colmenar, J. M., & Duarte, A. (2019a). A variable neighborhood search approach for the hamiltonian p-median problem. *Applied Soft Computing*, *80*, 603 – 616.

Herrán, A., Colmenar, J. M., & Duarte, A. (2019b). A variable neighborhood search approach for the vertex bisection problem. *Information Sciences*, *476*, 1 – 18.

Herrán, A., Colmenar, J. M., & Duarte, A. (2020a). An efficient metaheuristic for the k-page crossing number minimization problem. *Knowledge-Based Systems*, *207*, 106352.

Herrán, A., Colmenar, J. M., Martí, R., & Duarte, A. (2020b). A parallel variable neighborhood search approach for the obnoxious p-median problem. *International Transactions in Operational Research*, *27*, 336–360.

Hungerländer, P. (2014). A semidefinite optimization approach to the parallel row ordering problem. *Institut für Mathematik, Alpen-Adria-Universität Klagenfurt, Klagenfurt, Austria, Rep. TR-ARUK-MO-14-05*, .

Kalita, Z., Datta, D., & Palubeckis, G. (2019). Bi-objective corridor allocation problem using a permutation-based genetic algorithm hybridized with a local search technique. *Soft Comput.*, *23*, 961986.

Loiola, E. M., de Abreu, N. M. M., Boaventura-Netto, P. O., Hahn, P., & Querido, T. (2007). A survey for the quadratic assignment problem. *European Journal of Operational Research*, *176*, 657 – 690.

Mladenović, N., & Hansen, P. (1997). Variable neighborhood search. *Computers & Operations Research*, *24*, 1097 – 1100.

Wess, B., & Zeitlhofer, T. (2004). On the phase coupling problem between data memory layout generation and address pointer assignment. In *International Workshop on Software and Compilers for Embedded Systems* (pp. 152–166). Springer.

Yang, T., & Peters, B. A. (1997). A spine layout design method for semiconductor fabrication facilities containing automated material-handling systems. *International Journal of Operations & Production Management*, .