INTERNATIONAL TRANSACTIONS IN OPERATIONAL RESEARCH

WILEY

INTERNATIONAL TRANSACTIONS IN OPERATIONAL RESEARCH

Intl. Trans. in Op. Res. 24 (2017) 635–662
DOI: 10.1111/itor.12309

Parallel variable neighborhood search for the min–max order batching problem

Borja Menéndez[a], Eduardo G. Pardo[b], Jesús Sánchez-Oro[a] and Abraham Duarte[a]

[a]*Dpto. Informática y Estadística, Universidad Rey Juan Carlos, C/Tulipán s/n, Móstoles, Madrid, Spain*
[b]*Dpto. Sistemas Informáticos, Universidad Politécnica de Madrid, Ctra. Valencia, Km. 7, Madrid, Spain*
E-mail: borja.menendez@urjc.es [Menéndez]; eduardo.pardo@upm.es [Pardo]; jesus.sanchezoro@urjc.es [Sánchez-Oro]; abraham.duarte@urjc.es [Duarte]

Received 28 March 2015; received in revised form 19 January 2016; accepted 19 April 2016

Abstract

Warehousing is a key part of supply chain management. It primarily focuses on controlling the movement and storage of materials within a warehouse and processing the associated transactions, including shipping, receiving, and picking. From the tactical point of view, the main decision is the storage policy, that is, to decide where each product should be located. Every day a warehouse receives several orders from its customers. Each order consists of a list of one or more items that have to be retrieved from the warehouse and shipped to a specific customer. Thus, items must be collected by a warehouse operator. We focus on situations in which several orders are put together into batches, satisfying a fixed capacity constraint. Then, each batch is assigned to an operator, who retrieves all the items included in those orders grouped into the corresponding batch in a single tour. The objective is then to minimize the maximum retrieving time for any batch. In this paper, we propose a parallel variable neighborhood search algorithm to tackle the so-called min–max order batching problem. We additionally compare this parallel procedure with the best previous approach. Computational results show the superiority of our proposal, confirmed with statistical tests.

Keywords: min–max order batching problem; parallel variable neighborhood search; general variable neighborhood search

1. Introduction

Warehouse management systems have become an essential part of the supply chain strategy. They mainly focus on moving and storing materials within a warehouse by performing different transactions (including shipping, receiving, and picking). The success of a warehouse management system strongly depends on how customer orders (containing a set of goods or items) are retrieved. This picking process may consume up to 60% of the total time of all labor activities in the warehouse (Drury, 1988), which can assume more than half of the total operating costs. It consists of taking

and collecting articles (items) in a specified quantity before shipment to satisfy the orders of the customers.

As it is well documented in the literature, reducing the travel time of picking operations improves order picking productivity. This is why batch strategies are used in warehouses and it is also one of the aims that makes companies invest into conveyor systems. Travel time can easily account up to 50% or even more of the time spent on order picking tours. Note that other time-consuming activities might include searching items, acceleration and deceleration of the picking cart, approximation to the shelves, etc. By combining orders into a single batch, the time spent traveling is greatly reduced. The smaller the order, the better the opportunity to combine multiple orders into a single batch.

Given a set of orders received in the warehouse, there are two basic order-picking strategies: "strict-order picking" and "order batching." In the first strategy, each picker collects all the items included in one order. Once he/she finishes, the picker continues with the second order and so on. In the order batching strategy, several orders are put together into batches. Then, each batch is assigned to a picker, who can retrieve the items of any order grouped into the assigned batch, and satisfy a capacity constraint (fixed by a maximum number of orders).

From an optimization point of view, we can identify two different problems: how to group orders into batches (batching) and how to design the corresponding routes to collect them (routing). According to De Koster et al. (1999a), if both problems (i.e., batching and routing) are simultaneously considered, the associated benefits can be considerably increased, reducing travel time more than 35%.

In this paper, we deal with an optimization problem that involves these two main actions: (a) grouping the orders into feasible batches and (b) retrieving the items in each order from their location, satisfying some constraints. In particular, the optimization problem addressed in this paper is known as min–max order batching problem (OBP), where the specific aim is to minimize the maximum retrieving time for all the generated batches.

We propose sequential and parallel algorithms based on the variable neighborhood search (VNS) methodology to address it. The remainder of the paper is organized as follows. In Section 2, we provide a literature review of the state of the art. In Section 3, we describe the problem. In Section 4, we present different strategies based on VNS to tackle the problem. The parallel VNS algorithm designed to address the min–max OBP is described in Section 5. The computational experiments and associated conclusions are presented in Sections 6 and 7, respectively.

2. Literature review

The aforementioned problems have been approached using different models. The OBP is probably the most studied variant. It consists of distributing the orders among an unconstrained number of batches such that the sum of the time of the picking tours (one per batch) is minimized. The OBP has been proved to be NP-hard for general instances, but it is solvable in polynomial time if each batch does not contain more than two orders (Gademann and Velde, 2005). Unfortunately, real warehouse instances do not usually fall into this category. Therefore, the OBP has been heuristically approached in the past years. For the sake of clarity, we restrict our revision to modern heuristic procedures. In particular, Hsu et al. (2005) proposed a genetic algorithm for the OBP considering any warehouse layout. Later, Albareda-Sambola et al. (2009) proposed a variable neighborhood descent (VND)

algorithm that considered six different neighborhood structures. Henn et al. (2010) proposed two procedures. The first approach is an iterated local search (ILS) algorithm that consists of a local search procedure, which considers two neighborhoods, and a complex perturbation procedure. The second method is a straightforward implementation of a rank-based ant system. Henn and Wäscher (2012) improved previous results by proposing two metaheuristic methods. The first one is tabu search algorithm that constructs the initial solution with a classical method (Clarke and Wright, 1964; Gibson and Sharp, 1992) and improves it with a nested exploration of two different neighborhoods. The second procedure is an attribute-based hill climbing (ABHC) method with two genuine attribute sets. The most recent approach to tackle the OBP was presented in Öncan (2015), where the author proposed several mixed integer linear programming (MILP) formulations to solve the problem. More precisely, the difference among methods relies on the idea of using different routing strategies to evaluate the solution. Additionally, the author proposed an ILS with a tabu thresholding method that considers three different neighborhoods.

The online rescheduling OBP (OR-OBP) is a dynamic extension of the OBP (Rubrico et al., 2011). In this variant, new orders are randomly added to the list of orders to be picked even while the static orders are being retrieved. The number of batches is not limited and the objective function is again the minimization of the total travel time. It is assumed that static and dynamic orders have the same priority and all of them must be picked. Additionally, a dynamic order cannot be inserted into a route currently being traversed by a picker. However, it can be added to a nonretrieved batch, while capacity constraint is not violated. The OR-OBP has been recently (and simultaneously) studied in Rubrico et al. (2011) and Henn (2012). In the former, the authors proposed a steepest descent algorithm and a problem-dependent heuristic. Henn (2012) studied a slightly different variant (online OBP), where there are only dynamic orders. He described a modified version of ILS (previously used in the context of OBP) for this variant. More recently, Chen et al. (2016) studied a more constrained version of this problem. Specifically, the authors considered the online version with only dynamic orders but avoiding the congestion in the aisles (i.e., it is not allowed that two or more pickers traverse the same point in an aisle at the same time). Chen et al. (2016) proposed an ant colony optimization (ACO) algorithm to solve this variant in which each ant simulates the behavior of humans in a warehouse.

The order batching and sequencing problem (OBSP) considers on time retrievals from a warehouse to avoid production delays. In this variant, instead of determining the quality of a solution by means of the total travel time, the order batching has to be evaluated with respect to the tardiness of the corresponding orders (Elsayed et al., 1993). The tardiness is defined as the positive difference between the collecting time of an order and its due date. If an order is collected before its due date, the tardiness associated to that order is zero. Note that the orders of a given batch have the same collecting time but some of them might satisfy a determined due date constraint and others might not. The goal of this problem is then to minimize the sum of the tardiness associated to each order. The OBSP has been recently studied in Henn and Schmid (2013), using a modified version of the ILS and ABHC methods proposed in Henn et al. (2010) and Henn and Wäscher (2012), respectively. Azadnia et al. (2013) proposed a hybrid approach based on combining a weighted association rule mining to determine the affinity among orders with respect to their due date, a standard genetic algorithm to produce a feasible batch configuration, and a different genetic algorithm to construct the routes to collect batches. The last attempt to address the problem has been driven by Chen et al. (2015). In this paper, the authors developed first, a genetic algorithm to search near-optimal

solutions of order batching and batch sequencing and then, an ACO algorithm that searches the shortest path for each batch.

Henn (2015) proposed a VNS method with eight different neighborhood structures for a problem similar to the OBSP but considering multiple pickers (MP). This new variant is known as OBSP with MP (OBSP-MP). A solution to the OBSP-MP can be interpreted as a set of batches and the corresponding sequences (one for each order picker). A warehouse operator collects the customer orders included in the first batch of his/her sequence. After returning to the depot he/she collects the customer orders in the second batch of his/her sequence and so on. The goal of this problem is to minimize the total tardiness.

Gademann et al. (2001) introduced the min–max OBP variant, which considers that a number of batches is collected simultaneously by a group of pickers. It is usually denominated as "wave picking operation." In the context of warehousing, these operations are applied when the set of orders to retrieve is large and, thus, the collecting time is important. Min–max OBP assumes that each picker collects the items of one batch and all the pickers start their routes at the same time. The objective function in this variant is to minimize the maximum retrieving time of any of the batches. This variant was exactly solved by Gademann et al. (2001) with a branch-and-bound (B&B) algorithm with four different lower bounds and a straightforward heuristic for the upper bound.

3. Problem description

The min–max OBP consists of a combination of two different tasks: batching and routing. In particular, the warehouses considered in this paper have parallel aisles of equal length connected by two cross aisles (one at the front and one at the back). They also have a depot located in either the front or the back cross aisle. In Fig. 1, we illustrate an example of a warehouse layout with five parallel aisles, two cross aisles, and a depot located at the left bottom corner.

In this problem, the warehouse receives orders and each one must be retrieved by a single picker. An order consists of one or more order lines, where each one determines the number of items of a given product that must be collected. The products are stored in the warehouse on different picking positions. A position might have one or more levels organized in different shelves.

A picker retrieves the items in a given batch by following a specific route that starts and finishes at the depot. The time employed in performing that route is known as "retrieving time" of a batch. It is computed as the sum of the "travel time" plus the "extraction time" that includes, as mentioned above, the time needed to localize the item and to extract it from its picking location, the associated time employed in accelerating/decelerating the picking cart, etc. In this case, the extraction time is considered as one unit per item retrieved. Note that the travel time is calculated by means of a routing algorithm.

Let n be the number of orders, m the number of batches, and $b = \lceil \frac{n}{m} \rceil$ the maximum number of orders per batch. The number of orders that can be held into a batch is determined by its capacity (i.e., it is assumed that the order weight/size/volume is one unit).

As mentioned earlier, the routing problem determines the value of the retrieving time. It can be exactly solved in polynomial time (see Ratliff and Rosenthal, 1983). They presented a dynamic programming algorithm that obtains a route by considering only six possible configurations of traversing an aisle. Similarly, there are five different ways of going through two consecutive aisles

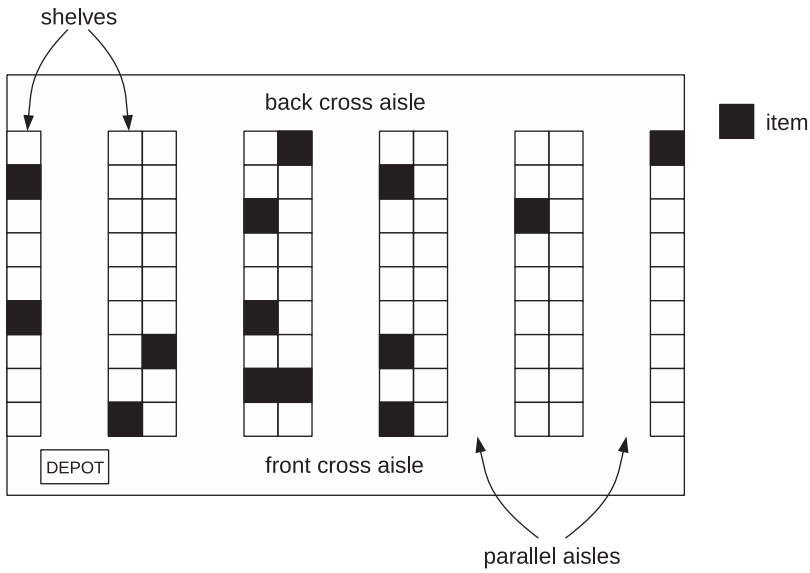

Fig. 1. Warehouse layout.

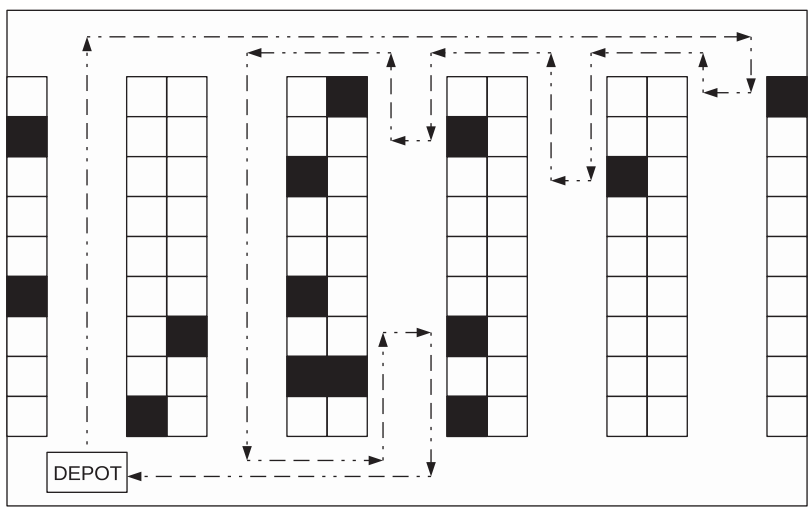

Fig. 2. Optimal route.

(using either the front or back cross aisles). Thus, determining the optimal route is polynomially solvable in $\mathcal{O}(n_a + n_i)$ time, where n_a is the number of aisles and n_i is the number of items to be retrieved.

However, these routes might be hard to follow for human pickers, since they could imply to move forward and backward in the same cross aisle. In Fig. 2, we show an example of how a picker collects the items of a particular batch following this approach. The picking locations that contain a required item are represented by black squares. Note that the route must reach the position of

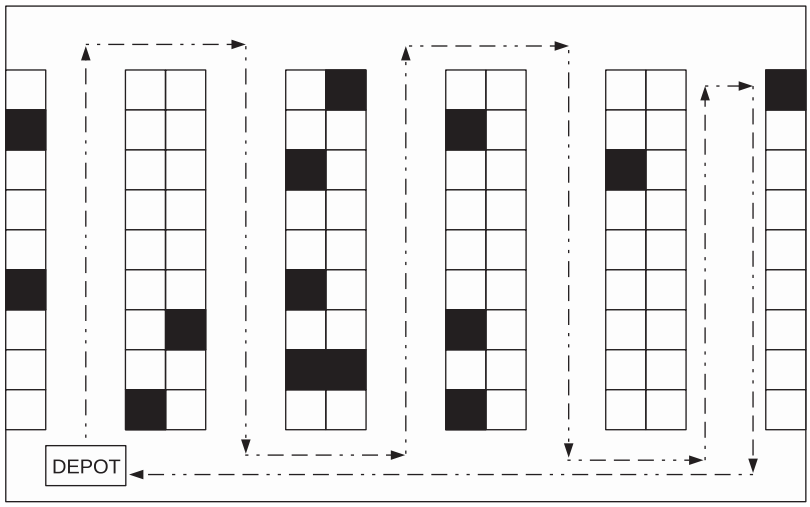

Fig. 3. S-shape strategy.

all black squares. The retrieving time is computed as the time needed to perform this route and collect all the required items. As we can see in this example, the picker must perform a U-turn in the front cross aisle. Therefore, this kind of solutions is usually discarded in real situations, resorting to heuristic strategies.

The easiest heuristic routing strategy for warehouse workers is the S-shape or traversal strategy (Goetschalckx and Ratliff, 1988). It consists of traversing an aisle, which contains at least one item to collect, from the front cross aisle to the back cross aisle (or the other way round). If the number of parallel aisles is odd, the last aisle is traversed until the farthest item from the front cross aisle. In Fig. 3, we show an example of how a picker collects the items of a particular batch. Then, the S-shape route must go through those aisles. Note that the last aisle is not fully traversed, since the number of aisles is odd. The retrieving time is computed as the time needed to perform this route and collect all the required items.

The largest gap strategy (De Koster et al., 1999b) is based on the idea of gap, which is defined in a parallel aisle as the distance between two consecutive items to be retrieved within the aisle; the distance between the front cross aisle and nearest item in the parallel aisle; or the distance between the back cross aisle and nearest item in the parallel aisle. The largest gap distance of a parallel aisle is the maximum of the previously defined distances. The largest gap strategy avoids to traverse the largest gap distance, performing a U-turn when the picker reaches the position of the item where the largest gap distance starts. In this routing strategy, the picker fully traverses both, the first and last aisles that contain required items. The rest of the parallel aisles are partially traversed since the picker avoids to traverse the largest gap distance. In Fig. 4, we illustrate an example of a batch retrieving, where the required items are again represented by black squares. As we can observe in this figure, the first and last aisles are fully traversed. The second and third aisles are entered and exited from the back cross aisle and front cross aisle (performing two U-turns). However, the fourth aisle is only entered and exited from the back cross aisle (performing only one U-turn).

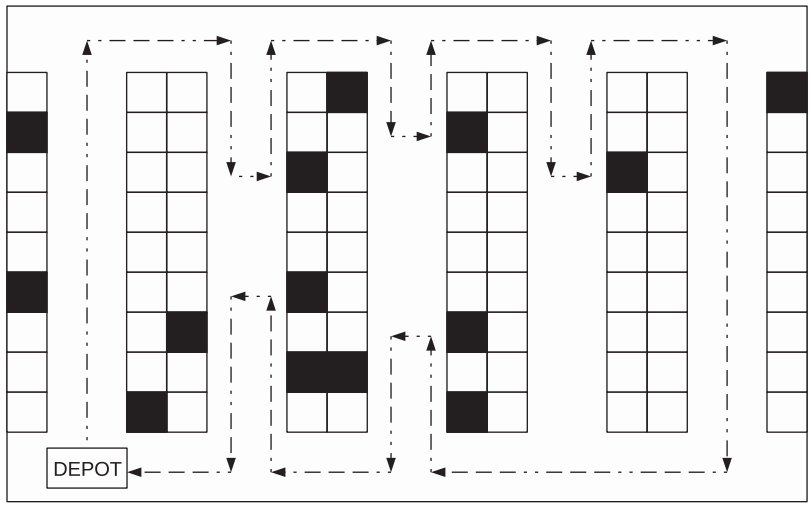

Fig. 4. Largest gap strategy.

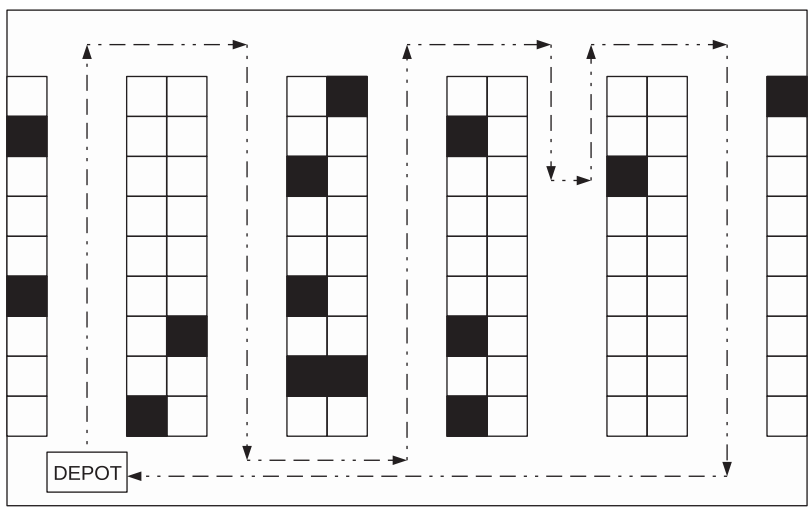

Fig. 5. Combined strategy.

The combined strategy was originally introduced in Roodbergen and De Koster (2001) as a combination of the two previously defined strategies. Afterwards, Albareda-Sambola et al. (2009) proposed a more elaborated version of this routing algorithm. In particular, it determines whether it is better to traverse an aisle with the S-shape or with the largest gap strategy. More precisely, after collecting all items of one aisle, the algorithm decides whether to go to the rear end of an aisle (S-shape) or to return to the front end (largest gap). These two alternatives are compared to each other, selecting the one that results in the shortest route. Thus, after leaving an aisle, the algorithm evaluates the same two alternatives for the next one. This means that there are always two possible routes to follow. In Fig. 5, we show an example of a route obtained by the combined strategy where

the first, second, third, and fifth aisles are traversed with the S-shape strategy and the fourth aisle is traversed with the largest gap strategy.

In general, the implementation of the combined strategy is considerably more complex than the other two. However, it usually produces better outcomes (Albareda-Sambola et al., 2009). In this paper, we then select the combined strategy among the heuristic approaches to be compared to the exact method (Ratliff and Rosenthal, 1983) in order to construct the routes of the picker. This comparison could help to analyze the independence of the batching and routing strategies.

4. VNS methodology

VNS is a metaheuristic introduced by Mladenović and Hansen (1997), which exploits the idea of neighborhood change in a systematic way, both descending to a local optimum and escaping from the basin of attraction of that local optimum. The original metaheuristic has been widely evolved with many extensions. For instance, VND explores the neighborhood in a deterministic way by using several local search procedures; reduced VNS (RVNS) explores solutions at random in each neighborhood by perturbing the solutions with the shaking procedure; basic VNS (BVNS) combines deterministic and random exploration of the neighborhoods.

In this paper, we propose a general VNS (GVNS) for the min–max OBP. This scheme was first used in Caporossi et al. (1999), although they had no name yet. The pseudocode of this algorithm is shown in Algorithm 1. This scheme has three input parameters, namely, the initial solution (S), largest neighborhood to be explored (k_{max}), and maximum computing time (t_{max}). The influence of the parameter k_{max} on the search is studied in Section 6. The t_{max} is set in order to match the computing time of previous approaches in the state of the art.

The construction of the initial solution does not belong to the GVNS scheme. In fact, the initial solution might be constructed at random, as it is usual in many papers by the VNS community. However, as it is well documented in the related literature (see Duarte et al., 2012; Lozano et al., 2012; Sánchez-Oro et al., 2014, 2015), a more elaborated constructive procedure might improve considerably the quality of the best solution found. In Section 4.1, we describe the constructive procedures proposed for the min–max OBP.

The GVNS algorithm starts by initializing k to the first neighborhood (step 3) and then, it enters the main loop (steps 5–7). First, the current solution is perturbed, in the shake procedure, by applying a random move in the kth neighborhood (see Section 4.2). Then, the obtained solution is improved within a VND procedure that considers two different neighborhoods (see Section 4.3). The VND is described in detail in Section 4.4. Finally, we use a standard implementation of neighborhood change method. In particular, this procedure determines which neighborhood is the next one to be explored. If the perturbed and improved solution (S'') outperforms S, the GVNS method considers S'' as the new incumbent solution ($S \leftarrow S''$) and resets the search to the first neighborhood ($k \leftarrow 1$). Otherwise, the current solution is not updated, but a larger neighborhood is explored ($k \leftarrow k + 1$). These three steps are repeated until the largest neighborhood (k_{max}) is explored without finding an improvement. Then if the maximum computing time (t_{max}) has not been exceeded, the GVNS performs a new iteration; otherwise, it returns the best solution found.

Algorithm 1. GVNS algorithm

```
1: procedure GVNS(S, k_max, t_max)
2:     repeat
3:         k ← 1
4:         repeat
5:             S' ← Shake(S, k)
6:             S'' ← VND(S')
7:             NeighborhoodChange(S, S'', k)
8:         until k > k_max
9:         t ← CPUTime()
10:    until t ≥ t_max
11:    return S
12: end procedure
```

4.1. Constructive methods

We propose two different greedy constructive procedures. Both of these procedures fill batches sequentially, that is, they start filling the first batch, then the second batch, and so on. Algorithm 2 shows the pseudocode of the first constructive procedure, denominated C1. It begins by initializing the solution S to the empty set (step 2). Then, we define $\mathcal{O}$ as the set of orders not included yet in the solution (step 3). Note that we do not have a complete solution until $\mathcal{O}$ is empty. Once these variables are initialized, the procedure greedily selects the orders to be included in each batch (steps 4–12). In particular, batch B_i is initialized in step 5. Then, while the size constraint is not satisfied (steps 6–10), the greedy function, denoted as g_1, determines the best order to be included in the solution under construction. Specifically, C1 selects the order $O^\star$ that minimizes the retrieving time of the current batch (step 7). When an order has been selected, it is included in the corresponding batch (step 8) and removed from the set $\mathcal{O}$ (step 9). When the batch B_i is filled, it is added to S (step 11) and C1 performs a new iteration. The constructive procedure ends when all orders have been assigned to one of the m available batches, returning the solution S.

The second constructive procedure, denominated C2, considers a different greedy function, g_2. Specifically, instead of selecting the order that minimizes the retrieving time of the batch B_i, C2 chooses the order that maximizes the relative number of shared items between the corresponding order and the items already in B_i. Let us illustrate this new greedy function with an example. Suppose that the batch B_i already has the orders O_a and O_b, and we want to evaluate the greedy value when we include order O_c. Then, this value is computed as

$$g_2(B_i, O_c) = \frac{|O_a \cap O_c| + |O_b \cap O_c| - |O_a \cap O_b \cap O_c|}{|O_a \cup O_b \cup O_c|}.$$

Algorithm 2. Scheme for the constructive algorithms

1: **function** C1
2: $S \leftarrow \emptyset$
3: $\mathcal{O} \leftarrow \{O_1, O_2, \ldots, O_n\}$
4: **for** $i = 1$ **to** m **do**
5: $B_i \leftarrow \emptyset$
6: **while** $|B_i| < b$ and $\mathcal{O} \neq \emptyset$ **do**
7: $O^\star \leftarrow \arg\min_{O \in \mathcal{O}} g_1(B_i, O)$
8: $B_i \leftarrow B_i \cup \{O^\star\}$
9: $\mathcal{O} \leftarrow \mathcal{O} \backslash \{O^\star\}$
10: **end while**
11: $S \leftarrow S \cup B_i$
12: **end for**
13: **return** S
14: **end function**

Note that we subtract the intersection of O_a, O_b, and O_c to avoid double counting those items that are simultaneously in the three orders. We do not provide the pseudocode of this procedure, since it is completely equivalent to the procedure presented in Algorithm 2, but replacing the greedy function g_1 with g_2 and min-operator with max-operator (see step 7 in Algorithm 2).

4.2. Shake strategy

The shake stage is usually introduced in VNS as an effective strategy to escape from a basin of attraction. Given a solution S, the shake procedure randomly generates a solution S' by applying k moves to S. In general, it is said that solution S' is in the neighborhood $N_k(S)$. More precisely, $N_k(S)$ contains the set of solutions that can be reached by applying k consecutive moves.

There are some optimization problems where the topology of the search space is so steep that a straightforward implementation of the shake procedure is not enough to escape from a basin of attraction, and we require more aggressive strategies. As we will empirically show in the computational experience, the min–max OBP falls into this category (i.e., the depth and height of some local maxima and minima are so large that it is not always possible to escape from them by simply shaking the current solution). We overcome this situation using a more elaborated perturbation schema (involving four different orders allocated in four different batches). This move is inspired in the ejection chain methodology (Glover, 1992). Figure 6 illustrates how this move works. It considers four different batches, namely, B_a, B_b, B_c, B_d and one order per batch (O_a, O_b, O_c, O_d, respectively). The proposed shake procedure requires that, at least, one of the involved batches presents the largest retrieving time since that batch determines the value of the objective function.

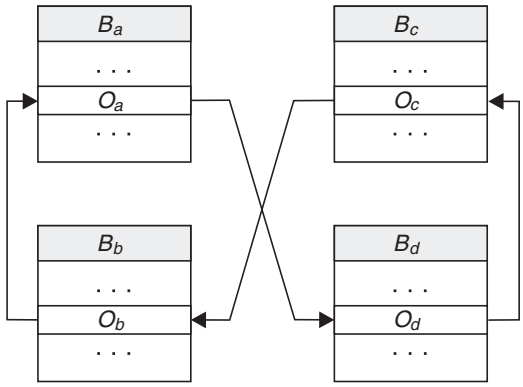

Fig. 6. Shake procedure.

The other three batches are selected at random. The four orders involved in the shake strategy are also selected at random from its corresponding batch.

The shake procedure starts by selecting one order, $O_a \in B_a$, removing it from its current batch and inserting it in batch B_d. In order to make room for O_a, the order O_d is removed from its current batch (B_d) and it is inserted in B_c, producing the ejection of order O_c from B_c to B_b. Finally, order O_b is moved from B_b to B_a finishing the chain. Note that this chain represents a perturbation within the shake procedure (i.e., $k = 1$). Therefore, $k = 2$ implies two different chains of moves, and so on. In this case, the batch B_a is always the batch with the largest retrieving time. Otherwise, the shake strategy does not produce the desired effect.

4.3. Neighborhood structures

A solution to the min–max OBP is represented as a list of m batches (i.e., $S = \{B_1, B_2, \ldots, B_m\}$), where each batch B_j contains b orders, and each order is formed by an unfixed number of items that must be retrieved. According to this solution representation, the swap move, denoted by $Swap(S, O_i, B_j, O_k, B_l)$, produces a new solution S' starting from the solution S, where the order O_i is removed from its current batch (B_j) and inserted in batch B_l. Simultaneously, the order O_k is removed from its batch B_l and inserted in B_j. Considering this move, the neighborhood associated to the solution S is

$$N_{\text{Swap-1}}(S) = \{S' \leftarrow Swap(S, O_i, B_j, O_k, B_l) \quad \text{with } 1 \le i, k \le n, \quad 1 \le j, l \le m, \quad i \ne k, \quad \text{and} \quad j \ne l\}.$$

The second considered neighborhood is based on a composed move that consists of applying the swap move two consecutive times. This move is implemented as $Swap(Swap(S, O_i, B_j, O_k, B_l), O_p, B_l, O_q, B_r)$. Specifically, it considers four orders (O_i, O_k, O_p, and O_q) allocated in three different batches (i.e., $O_i \in B_j$, $O_k \in B_l$, $O_p \in B_l$, and $O_q \in B_r$). This move swaps the orders O_i and O_k between their respective batches (B_j and B_l) and then, swaps orders O_p

and O_q between their respective batches (B_l and B_r). Note that O_i and O_p might be the same order since the inner *Swap*-move is always performed before the outer *Swap*-move. The neighborhood of the solution S generated by this move is

$$N_{\text{Swap-2}}(S) = \{S' \leftarrow Swap(Swap(S, O_i, B_j, O_k, B_l), O_p, B_l, O_q, B_r) \text{ with } 1 \le i, k, p, q \le n, \quad 1 \le j, l, r \le m, \quad i \ne k, \quad p \ne q, \quad i \ne q, \quad j \ne l, \quad l \ne r, \quad \text{and} \quad r \ne j\}.$$

The complexity of exploring the whole neighborhood (the worst case) is cubic with respect to the number of batches, i.e., $|N_{\text{Swap-2}}(S)| = O(m^3)$. Therefore, if it is exhaustively explored, the associated computing time could be considerably enlarged. In order to reduce this time, we fix B_j to be the batch with the largest retrieving time. This simplification reduces $N_{\text{Swap-2}}(S)$ from cubic to quadratic size in terms of the number of batches.

4.4. Improvement methods

Local search methods are likely the oldest and simplest heuristic methods used to improve solutions. Starting from a given feasible solution, these procedures explore a determined neighborhood in each iteration, replacing the current solution if a neighbor improves the objective function. The search ends when all neighbor solutions are worse (i.e., larger objective function value in a minimization problem), meaning that a local optimum is found.

There exist two typical strategies to explore the corresponding neighborhood: best improvement and first improvement. In the former, the associated neighborhood is completely explored by a fully deterministic procedure, performing the best associated move. In the latter, it tries to avoid the scanning of the whole neighborhood by performing the first improving move encountered during the exploration of the corresponding neighborhood. In general, iterations performed in the first improvement strategy are more efficient than iterations in the best improvement strategy, since the former only evaluates part of the neighborhood, while the latter explores it completely. On the other hand, the improvement obtained in the first improvement strategy is typically smaller than the improvement achieved by the best improvement strategy, requiring in general more iterations to obtain the local optimum.

A straightforward implementation of the best improvement local search, based on swap moves, implies to explore a relatively large neighborhood. Specifically, in $N_{\text{Swap-1}}$, the swap move evaluates for each batch and each order its potential swapping with the remaining batches and orders. Therefore, the size of this neighborhood is $m^2 \times b^2$. As it is pointed out by Hoos and Stützle (2004), the best improvement strategy is usually more adequate to perform efficient caching and updating mechanisms, which allows the search to efficiently explore the neighborhood. Considering the ideas described in Clarke and Wright (1964) about saving matrices, we propose an efficient local search method, denoted as LS1, to explore $N_{\text{Swap-1}}$. It is based on a nonsymmetric $n \times n$ matrix, where each element located in the row i and column j stores the increment (positive value) or decrement (negative value) of the retrieving time of each batch if we swap the order O_i (in one batch) with the order O_j (in a different batch). Table 1 shows an example of this matrix for a problem with six batches

Table 1
Difference of retrieving time of orders before a swap move

		B_1				B_2				B_3				B_4				B_5				B_6			
		O_1	O_2	O_3	O_4	O_5	O_6	O_7	O_8	O_9	O_{10}	O_{11}	O_{12}	O_{13}	O_{14}	O_{15}	O_{16}	O_{17}	O_{18}	O_{19}	O_{20}	O_{21}	O_{22}	O_{23}	O_{24}
B_1	O_1	0	0	0	0	5	0	−9	5	14	14	4	−7	6	4	−2	10	−4	−6	−4	8	−8	−6	−5	4
	O_2	0	0	0	0	−3	−5	2	−1	−11	14	15	10	10	−8	14	−6	12	−4	4	3	−5	8	−7	10
	O_3	0	0	0	0	6	6	−5	−6	1	9	−10	−5	4	−6	13	7	−7	3	−5	2	−2	1	6	4
	O_4	0	0	0	0	−6	−1	13	−7	−1	9	12	−1	−6	9	9	−1	−3	9	15	−9	11	13	−4	8
B_2	O_5	−10	−5	13	−6	0	0	0	0	2	11	1	3	6	−6	13	6	9	−4	−2	−7	0	0	15	10
	O_6	−5	2	−1	1	0	0	0	0	−6	−10	7	−4	4	11	7	7	−5	8	11	15	14	13	12	14
	O_7	−2	0	−5	−5	0	0	0	0	13	2	1	−7	1	10	4	−6	4	−8	15	−5	5	6	−4	2
	O_8	−1	6	−5	−5	0	0	0	0	−9	15	5	−8	−7	−3	14	−2	−3	9	−9	1	−8	3	13	11
B_3	O_9	5	1	10	15	1	7	2	3	0	0	0	0	10	−4	15	−6	7	15	15	12	7	10	11	5
	O_{10}	9	2	0	−2	4	−8	8	−7	0	0	0	0	7	8	4	−4	10	3	1	−1	−6	8	12	1
	O_{11}	−10	7	12	13	−8	8	12	8	0	0	0	0	1	13	13	3	1	11	−1	−4	−8	−2	0	10
	O_{12}	14	−5	−7	1	−10	−7	−2	1	0	0	0	0	−3	−1	6	15	−6	2	4	12	−6	−8	−8	15
B_4	O_{13}	6	14	−9	0	−6	−9	6	4	−6	11	0	0	0	0	0	0	5	−7	9	5	−4	−6	6	0
	O_{14}	0	12	14	0	4	15	−9	−8	14	8	−6	−10	0	0	0	0	11	−10	6	0	−5	10	−10	0
	O_{15}	−7	0	12	7	6	7	6	−10	13	−2	15	9	0	0	0	0	4	10	−8	14	1	3	2	15
	O_{16}	12	13	−5	−2	0	3	8	3	5	−2	8	4	0	0	0	0	2	8	−7	3	13	14	−9	0
B_5	O_{17}	5	−9	−5	−5	−9	11	7	−9	9	4	−3	−9	13	−7	−6	3	0	0	0	0	6	5	−8	7
	O_{18}	−3	−2	−6	15	−1	8	9	3	7	8	5	−1	10	13	−6	−7	0	0	0	0	0	−10	10	3
	O_{19}	−10	−1	9	15	7	−10	4	12	−3	11	−2	−7	2	1	−8	13	0	0	0	0	−6	−2	4	13
	O_{20}	−1	15	14	−1	−6	14	−3	0	−9	−2	−6	3	8	−9	11	7	0	0	0	0	−1	−6	−1	12
B_6	O_{21}	11	0	11	5	4	−3	−9	−9	−2	−3	0	15	−7	−9	8	15	6	−7	1	1	0	0	0	0
	O_{22}	−1	6	−2	14	2	15	0	−6	−10	14	12	13	12	15	4	8	10	−7	14	0	0	0	0	0
	O_{23}	−7	10	3	−6	7	−2	15	−2	7	−1	12	12	6	5	15	15	13	13	−3	−3	0	0	0	0
	O_{24}	−5	−10	−7	10	9	−10	−4	−8	15	10	15	−4	−6	12	2	6	4	13	−10	5	0	0	0	0

and four orders per batch. For the sake of simplicity, the orders are grouped together according to their batches and lexicographically numbered. Note that if we swap two orders belonging to the same batch, the retrieving time does not change. For example, as illustrated in Table 1, the swap of orders O_1, O_2 does not produce a change in the retrieving time of the batch. In other words, the corresponding value of row 1 (column 2; symmetrically, row 2 (column 1)) is zero. On the other hand, if we swap the order O_2 with the order O_9, we would obtain a decrement of 11 time units (see row 2 (column 9)) for the batch B_1 and, simultaneously, it would produce an increment of one time units for batch B_3 (see row 9 (column 2)). The potential improvement would be the sum of these two values. If it results in a positive value, the corresponding swap move would produce a deterioration of the current solution; otherwise it would be considered an improving move. In the example of swapping O_2 with O_9, the resulting value is $-11 + 1 = -10$, which means that this move eventually improves upon the current solution.

We consider an additional array with size n that stores, for each order, the best possible swap with another order. We do not show this array since it can be easily derived by computing the minimum value of each row of the matrix. The best improving move is then the minimum value of this array. In the example described above, the best available move is to swap O_2 with O_9, reducing the value of the retrieving time of B_1 in 11 time units.

Once we perform a swap move, the matrix must be updated. In particular, if we perform the best move, that is, $Swap(S, O_2, B_1, O_9, B_3)$, we only need to recalculate part of this matrix. In particular, we need to reevaluate the retrieving time of the batches involved in the corresponding move. In Table 2, we show the new matrix after performing the best move, where the modified values are highlighted in gray.

The second neighborhood $N_{\text{Swap-2}}$ is constructed by applying the swap move for two consecutive times, affecting three different batches. As it was mentioned above, one of these is set to the batch with the largest retrieving time. Therefore, the size of this neighborhood is $m^2 \times b^3$ in the worst case, which is even larger than $N_{\text{Swap-1}}$. This neighborhood can either be explored using the first or best improvement strategy. We select the former since $N_{\text{Swap-2}}$ is scanned within a VND procedure, where it first explores $N_{\text{Swap-1}}$ (until reaching a local optimum with respect to that neighborhood), and then $N_{\text{Swap-2}}$. If we explored $N_{\text{Swap-2}}$ with the best improvement strategy, we should use a matrix similar to the matrix described for $N_{\text{Swap-1}}$. Then, after performing a move in $N_{\text{Swap-1}}$, we should update both matrices (although the method does not perform a move in $N_{\text{Swap-2}}$). Even worse, once we get a local optimum with respect to $N_{\text{Swap-1}}$, the VND method explores $N_{\text{Swap-2}}$ where it is likely that, after a few iterations, it finds an improvement, returning again to $N_{\text{Swap-1}}$. Therefore, this matrix is not exhaustively exploited, but requires large effort to keep it updated. We have experimentally confirmed this fact. Therefore, the local search method, denominated LS2, responsible of scanning $N_{\text{Swap-2}}$, is based on the first improvement strategy.

4.5. *Alternative objective function*

The objective function value of the min–max OBP consists in minimizing a maximum value. Consequently, there may be many different solutions with the same objective function value, since it is only determined by the batch with the largest retrieving time. These problems are a challenge for local search based methods, since these are based on improving moves and most of these reduce

Table 2
Difference of retrieving time of orders after a swap move

		B_1				B_2				B_3				B_4				B_5				B_6			
		O_1	O_9	O_3	O_4	O_5	O_6	O_7	O_8	O_2	O_{10}	O_{11}	O_{12}	O_{13}	O_{14}	O_{15}	O_{16}	O_{17}	O_{18}	O_{19}	O_{20}	O_{21}	O_{22}	O_{23}	O_{24}
B_1	O_1	0	0	0	0	1	−9	14	−6	−9	7	5	2	−4	−6	0	−7	−3	0	11	13	0	−7	−9	11
	O_9	0	0	0	0	7	−5	6	15	11	−3	7	−6	9	−1	−8	1	0	10	8	−9	10	−5	−8	12
	O_3	0	0	0	0	−8	12	6	5	−3	−9	−3	0	8	−2	−4	4	−7	6	−4	−10	6	2	7	−9
	O_4	0	0	0	0	−9	−7	10	11	8	8	12	13	−5	−8	3	−2	13	7	2	5	−6	−1	1	1
B_2	O_5	−6	14	−1	−8	0	0	0	0	−8	−1	5	−9	6	−6	13	6	9	−4	−2	−7	0	0	15	10
	O_6	3	10	12	13	0	0	0	0	10	−9	6	−4	4	11	7	7	−5	8	11	15	14	13	12	14
	O_7	−5	0	3	8	0	0	0	0	4	−1	7	12	1	10	4	−6	4	−8	15	−5	5	6	−4	2
	O_8	−7	9	−1	13	0	0	0	0	3	−2	−4	7	−7	−3	14	−2	−3	9	−9	1	−8	3	13	11
B_3	O_2	−8	−1	−3	−4	5	−1	−2	9	0	0	0	0	9	7	7	−4	0	14	6	9	0	−10	1	−5
	O_{10}	2	−6	−6	−4	−3	4	7	−6	0	0	0	0	12	1	−6	−7	11	−10	0	11	−8	−4	−4	−6
	O_{11}	1	3	−7	−9	11	−2	8	−7	0	0	0	0	−1	2	10	11	11	1	14	3	−2	3	8	−4
	O_{12}	7	14	−3	5	8	13	5	10	0	0	0	0	10	13	11	6	−7	10	−3	−1	3	2	13	6
B_4	O_{13}	4	−1	−8	−7	−6	−9	6	4	12	−7	7	14	0	0	0	0	5	−7	9	5	−4	−6	6	0
	O_{14}	−5	−8	14	5	4	15	−9	−8	8	7	8	−6	0	0	0	0	11	−10	6	0	−5	10	−10	0
	O_{15}	14	−6	7	14	6	7	6	−9	11	−2	15	3	0	0	0	0	4	10	−8	14	1	3	2	15
	O_{16}	9	2	11	0	0	3	8	3	15	−5	−8	3	0	0	0	0	2	8	−7	3	13	14	−9	0
B_5	O_{17}	7	−7	−7	9	−9	11	7	−9	12	2	10	11	13	−7	−6	3	0	0	0	0	6	5	−8	7
	O_{18}	5	6	−6	−8	−1	8	9	3	11	5	−1	−5	10	13	−6	−7	0	0	0	0	0	−10	10	3
	O_{19}	5	14	10	7	7	−10	4	12	4	9	−1	10	2	1	−8	13	0	0	0	0	−6	−2	4	13
	O_{20}	−2	11	2	11	−6	14	−3	0	8	−2	13	2	8	−9	11	7	0	0	0	0	−1	−6	−1	12
B_6	O_{21}	−7	−9	−6	−9	4	−3	−9	−9	9	9	15	0	−7	−9	8	15	6	−7	1	1	0	0	0	0
	O_{22}	13	10	−4	1	2	15	0	−6	14	−7	−7	13	12	15	4	8	10	−7	14	0	0	0	0	0
	O_{23}	12	−8	4	−9	7	−2	15	−2	4	−7	−2	−6	6	5	15	15	13	13	−3	−3	0	0	0	0
	O_{24}	10	3	3	−9	9	−10	−4	−8	6	−2	7	10	−6	12	2	6	4	13	−10	5	0	0	0	0

the retrieving time of a batch but not the time that determines the value of the objective function. Therefore, the value of the objective function is not modified.

Let $S = \{B_1, B_2, \ldots, B_m\}$ be a solution of the min–max OBP. Without loss of generality, we assume that the batches are sorted in a descending order of the corresponding retrieving time (i.e., $t_{B_1} \geq t_{B_2}, \cdots, \geq t_{B_m}$). Then, the value of solution S is t_{B_1}. It is not possible to improve the current objective function by performing a move without considering B_1. To overcome the lack of information provided by the move value in terms of the objective function, we extend here the meaning of "improving."

Let t_{B_j} and t_{B_l} be the retrieving times of batches B_j and B_l. Symmetrically, $t_{B'_j}$ and $t_{B'_l}$ are the retrieving times of B'_j and B'_l. A swap move, $Swap(S, O_i, B_j, O_k, B_l)$, produces a new solution $S' = \{B'_1, B'_2, \ldots, B'_m\}$, where

$$
\begin{aligned}
t_{B_s} &= t_{B'_s}, \quad s \neq j, s \neq l \\
B'_j &= B_j \setminus O_i \cup O_k \\
B'_l &= B_l \setminus O_k \cup O_i.
\end{aligned}
$$

In order to evaluate the quality of this move, we identify

$$
\begin{aligned}
t_{max} &= \max\{t_{B_j}, t_{B_l}\} \\
t_{min} &= \min\{t_{B_j}, t_{B_l}\} \\
t'_{max} &= \max\{t_{B'_j}, t_{B'_l}\} \\
t'_{min} &= \min\{t_{B'_j}, t_{B'_l}\}.
\end{aligned}
$$

Then, we assume that the swap move improves the current solution if $t'_{max} < t_{max}$ or, alternatively, $t'_{max} = t_{max}$ and $t'_{min} < t_{min}$. Let us illustrate it with an example. Suppose that the retrieving time of batches B_j and B_l are $t_{B_j} = 10$ and $t_{B_l} = 2$, respectively. Let us assume that after performing a move, the new retrieving times are $t'_{B_j} = 4$ and $t'_{B_l} = 9$. Then, although the retrieving time of B'_l has been considerably deteriorated (from 2 to 9), the corresponding move is accepted since the largest retrieving time has been reduced (from 10 to 9).

5. Parallel VNS

Nowadays, as a result of the evolution of computer architectures, modern computers are able to execute different programs simultaneously, since they have several processors. Computer scientists have been using this capability to increase the performance of their programs. This fact is particularly effective in the case of heuristics and metaheuristics (see Alba, 2005; Talbi, 2009). However, to make the most of the multiprocessor architecture, algorithms must be redesigned to be adapted to the specific architecture.

There exists several parallelization technologies that are suitable for implementing parallel algorithms (threads, OpenMP, CUDA, etc.). We refer the reader to Cook (2012), Gao et al. (2008),

Oaks and Wong (2004) for some tutorials on parallel programming. In this paper, we focus on the use of threads. In programming languages, a thread is defined as a fragment of code that is independently executed in a processor. "Pthreads" and "Java threads" are considered the most representative tools. In particular, Pthreads (POSIX threads) was defined in the mid-1990s as an effort to provide a unified set of C library routines in order to make multithreaded programs portable. Java threads is a version of Pthreads for Java programming language, which offers the advantages of the portability inherent to Java programs. In addition, Java threads can be easily used to tackle task parallel applications. Therefore, we select this technology to implement our parallel algorithms.

The parallelization of a metaheuristic must be intended to either reduce the computing time of the sequential algorithm or to explore a wider portion of the search space (García-López et al., 2002). The first step in parallelization is to identify which parts of the sequential algorithm can be redesigned to be executed in parallel efficiently. In this paper, we investigate an efficient parallelization of the VNS methodology (for some successful examples, see García-López et al., 2002; Crainic et al., 2004).

As far as we know, the first attempt to parallelize VNS was presented in García-López et al. (2002). Specifically, the authors proposed three different approaches: synchronous parallel VNS (SPVNS), replicated parallel VNS (RPVNS), and replicated shaking VNS (RSVNS). The idea behind the first approach, SPVNS, is to parallelize the local search method of a sequential VNS with the goal of reducing the computational time. This is mainly because the local search method is usually the most time-consuming part of the VNS algorithm. More precisely, SPVNS splits the execution of the local search in several threads. The second approach, RPVNS, explores a wider portion of the solution space using a multistart strategy. Specifically, RPVNS executes several VNS methods in parallel, each one in a different thread. The last variant, RSVNS, follows a classical master–slave scheme. In particular, the master executes the VNS, and each slave executes the shake and local search methods.

Later, Crainic et al. (2004) proposed a new parallel VNS approach called cooperative neighborhood VNS (CNVNS), which also uses the master–slave scheme. CNVNS considers the cooperative exploration of different neighborhoods by different threads. The master is responsible for maintaining, updating, and communicating the current overall best solution. It also initiates and terminates the algorithm executed in each thread. Each slave performs the exploration of a different neighborhood and communicates its local best solution to the master process. Then, the master communicates the best solution found to the slaves every time it is updated, making the slaves to continue the search from the new best solution.

To select the most suitable parallel VNS strategy for the current problem, it is essential to evaluate how each variant fits to the min–max OBP. The local search methods proposed for this problem are mainly sequential, that is, each iteration depends on the previous iteration. This fact makes the parallelization of the local search worthless. Therefore, SPVNS approach is not suitable for this problem. The RPVNS approach is well suited for VNS variants that use a multistart strategy. However, the algorithms proposed in this paper to construct solutions does not follow a multistart strategy. Then, the RPVNS approach is also discarded. The CNVNS is basically conceived for optimization problems where different type of moves are defined, determining different topologies of the search space. However, the moves associated to the min–max OBP are always based on swap moves (i.e., interchanging orders between batches). Therefore, the CNVNS cannot be easily adapted to this problem. This paper therefore focuses on the RSVNS methodology to redesign the

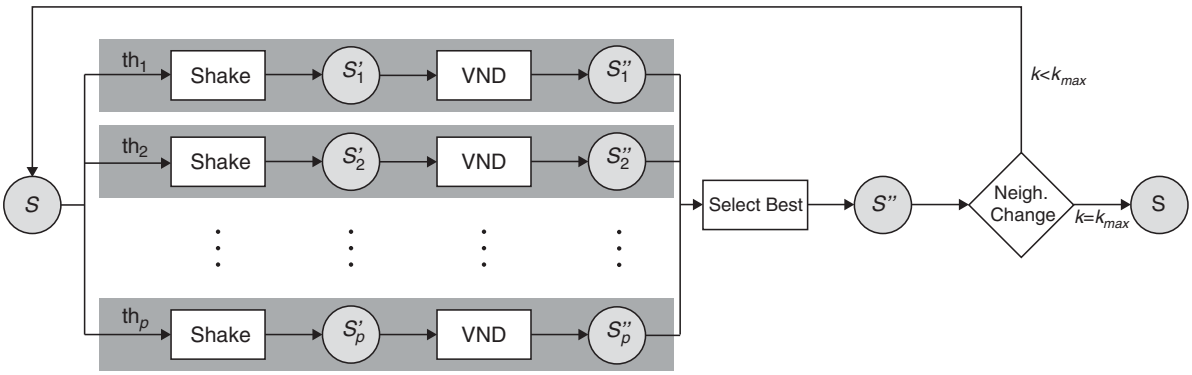

Fig. 7. Replicated shaking VNS.

general VNS described in Section 4 as a parallel procedure. This methodology allows the search to explore simultaneously p solutions (where p is the number of threads) in the current neighborhood. Therefore, in a single iteration, this variant is able to explore p solutions, while the sequential version explores only one. Note that if p processors are available, the computing time of the parallel and sequential versions should be similar, since the objective of RSVNS is not to reduce the computing time, but to explore a wider portion of the solution space.

In Fig. 7, we graphically show the proposed parallel procedure. In particular, the algorithm starts from an initial solution, S, constructed with the method proposed in Section 4.1, which becomes the current best solution. The algorithm then creates p threads, where each one perturbs the current solution S with the shake procedure, generating a new solution S'_i in the current neighborhood. It is important to remark that for each neighborhood, instead of generating only one solution (as the sequential implementation does), we generate p different shaken solutions, increasing the explored portion of the solution space.

Then, each solution is improved with the VND procedure described in Section 4.4, resulting in p local optima S''_i (with $1 \leq i \leq p$). The algorithm program waits until all the threads have finished the search with a barrier synchronization method. Then, the best solution, S'', among them is selected. Finally, the algorithm executes the *NeighborhoodChange* procedure defined in Section 4. It basically determines whether to perform a new parallel iteration ($k < k_{max}$) or not ($k = k_{max}$), returning the best solution found during the search. Note that this procedure is executed in the master thread (i.e., it is the only part of the VNS procedure not executed in parallel).

6. Computational experiments

In this section, we present the experiments performed to empirically study the influence of the proposed strategies and then to compare our best variant with the best algorithm identified in the state of the art (Gademann et al., 2001). We implemented our algorithms in Java 7, which were run on an Intel Core i7 with 2.5 GHz and 4 GB of RAM with Linux Mint Debian Edition 64 bit OS.

We consider two sets of instances previously used in the context of OBP. All of these are available at http://www.optsicom.es/mmobp/. We have divided the description of each set of instances into three different parts: warehouse layout, item distribution, and customer orders.

`Set HW` (Henn and Wäscher, 2012). This set contains 2560 instances whose main features are as follows:

- *Warehouse layout*: It consists of 900 storage locations, where each one stores a different article (item). The warehouse has 10 aisles with 90 storage locations each (45 on either side of each aisle). The length of each storage location is set to 1 length unit (LU). When the picker leaves an aisle, it is assumed that he/she moves 1 LU (from either the first or last storage location to the cross aisle). Finally, the picker spends 5 LU to move from the current aisle to the next aisle. The depot is located 1.5 LU away from the first storage location in the leftmost aisle.
- *Item distribution*: There are two different scenarios: (a) ABC distribution and (b) random distribution. In the first scenario, items can be grouped into three classes. The first scenario contains very demanded items (Class A), where 10% of the articles represent 52% of the demand. The second class contains items with medium demand (Class B), where another 30% of the articles accounts for 36% of the demand. Finally, Class C contains items with low demand and represents the final 60% of the articles that represents the 12% of the demand. Articles of Class A are stored in the first aisle, articles of Class B in the second, third, and fourth aisles, and articles of Class C in the remaining six aisles. Note that items are randomly located within a demand class. In the second scenario, items are randomly distributed among the storage locations.
- *Customer orders*: This set of instances considers four different sizes of orders, $n = \{40, 60, 80, 100\}$, where the number of items per order is uniformly distributed in $\{5, 6, \ldots, 25\}$. The capacity of the picking device C (defined as the maximum number of items that can be assigned to a batch) has been fixed to 30, 45, 60, and 75.

`Set AAMD` (Albareda-Sambola et al., 2009). This set contains 2400 instances whose main features are as follows:

- *Warehouse layout*: It considers four different warehouses: W1, W2, W3, and W4. The first warehouse has four parallel aisles with 60 storage locations (30 per side) and a total aisle length of 50 m. The distance between two consecutive aisles is 4.3 m and the picker speed is set to 0.6 m/second. W2 contains 10 aisles (20 storage locations on each side) with a length of 10 m. The separation between aisles is 2.4 m and the picker speed is also 0.6 m/second. The third warehouse has 25 aisles (25 storage locations per side) of length 50 m. The distance between two consecutive aisles is 5 m, and the picker speed 2 m/second. Finally, W4 has 12 aisles (16 storage locations per side) of 80 m length. The distance between two consecutive aisles is 15 m, and the picker speed is 1 m/second.
- *Item distribution*: This set of instances also considers an ABC distribution and random distribution. In this case, Class A contains 20% of the articles representing 80% of the demand. Class B contains 20% of the articles representing 10% of the demand. Finally, Class C contains the remaining 60% of the articles that represents 10% of the demand. Items belonging to each class are assigned to aisles according to these percentages. The random distribution is equivalent to the aforementioned distribution.

Table 3
Comparison between constructive methods

Constructive	Time (seconds)	Dev. (%)	#Best
C1	0.78	0.16	11
C2	0.01	2.93	6

- *Customer orders*: The order size for W1, W2, W3, and W4 is, respectively, $U(1, 7)$, $U(2, 10)$, $U(5, 25)$, and $U(1, 36)$, where U indicates a uniform distribution. The weight of each item is set to 1 for W1, W2, and W3. In the fourth warehouse, W4, the weight is randomly generated according to an uniform distribution $U(1, 3)$. Finally, the capacity of the picking device, C, is set to 12, 24, 150, and 80 for each warehouse, respectively.

The number of instances in the original sets is huge (totalizing 4960 instances). In particular, the `set HW` contains 64 groups of 40 similar instances. Similarly, the `set AAMD` has 80 groups of 30 similar instances. Therefore, we propose to use only one instance per group, selected at random, to ease future comparisons, reducing the original dataset from 4960 to 144 instances. We have observed that there are no significant differences between using the whole dataset or the reduced dataset. To this end, we have empirically tested this hypothesis by performing a previous experiment with simple heuristics (constructive procedure coupled with a local search). The obtained results were very similar, on average, in both datasets, which validates our original hypothesis.

In order to study the robustness of our proposal, we have considered different number of batches (i.e., $m = \{5, 10, 20, 25\}$). Considering that the instances reported in the literature establishes that the number of orders must be a multiple of the number of batches, we have then selected $m = \{5, 10, 20\}$ for `Set HW` and $m = \{5, 10, 25\}$ for `Set AAMD`. Therefore, the 144 instances previously selected becomes 432 instances, since we consider three values of m per instance.

6.1. Preliminary experimentation

In this section, we show the merit of the proposed search strategies by conducting several experiments with a representative subset of the instances selected, with different properties (10% of the whole set). We first study the performance of the proposed constructive procedures (see Section 4.1). Table 3 summarizes, for each method, the average execution time in seconds (Time (seconds)), the average percentage deviation with respect to the best results found in the experiment (Dev. (%)), and the number of times that each method matches the best sotion (#Best).

As expected, the computing time of these procedures is almost negligible (less than a second). In particular, C1 is considerably slower than C2. This difference in the computing time is mostly due to the fact that C1 executes a routing algorithm (see Section 3). Attending to the average deviation and the number of best solutions found, C1 seems to be better than C2. However, we have performed a Wilcoxon pairwise nonparametric test to certify whether the differences are significant or nonsignificant. The obtained p-value is 0.018, which indicates that there are statistically significant differences between both constructive methods, when considering a standard significance level of

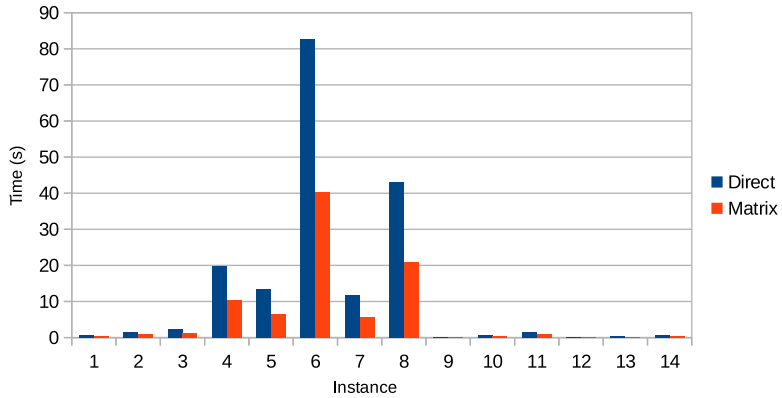

Fig. 8. Comparison of the execution time when using the "matrix-based" implementation or "direct-based" implementation in the local search.

Table 4
Comparison between LS1, LS2, and VND

Method	Time (seconds)	Dev. (%)	#Best
LS1	6.37	0.52	7
LS2	2.06	2.48	7
VND	9.62	0.12	12

$\alpha = 0.05$. We then select C1 as a constructive procedure since it presents better performance and the computing time is not relevant when comparing it with the execution time of the whole algorithm.

In the next experiment, we study the influence of the matrix of retrieving times in the performance of the local search, LS1 (see Section 4.4). In order to do so, we constructed solutions (with C1) and improved them with the two variants of LS1: using the cached information about retrieving times (matrix) or not (direct). In Fig. 8, we depict a bar diagram where the X-axis represents the instances in the preliminary set and Y-axis reports the computing time required to obtain a local optimum with both methods in the corresponding instance. As it is clearly shown in Fig. 8, the saving in computing time when using the "matrix-based" implementation is significant. Specifically, for these 14 instances "direct" needs 12.82 seconds on an average to obtain a local optimum, while "matrix" requires 6.37 seconds on an average to obtain the same local optimum, that is, two times faster. We then use the "matrix" variant for the remaining experiments.

In the third preliminary experiment, we compare the performance of the VND (see Section 4.4), with the local search procedures in isolation (i.e., LS1 and LS2). Typically, VND explores the neighborhoods from the smallest to the slowest and largest. Consequently, our VND first considers $N_{\text{Swap-1}}$ and then $N_{\text{Swap-2}}$. Results in Table 4 confirm that the VND procedure compares favorably with respect to simple local search methods. We have additionally confirmed this fact by conducting a Friedman nonparametric test, which ranks the algorithms according to their quality. The resulting p-value of 0.044 indicates that there exists statistical differences among the local search methods (considering $\alpha = 0.05$). Specifically, VND achieves the lowest deviation (0.12%) and a rank value of 1.64, compared to the two local search methods tested (0.52% with ranking 2.00 and 2.48%

Table 5
Influence of k_{max} on the performance of the GVNS

k_{max}	Time (seconds)	Dev. (%)	#Best
3	22.60	3.06	7
4	32.92	2.92	9
5	43.94	0.40	11
6	53.23	0.37	12
7	70.10	0.00	14

with ranking 2.36 for LS1 and LS2, respectively). It is worth mentioning that the results of LS2 in isolation seem to be quite bad. However, when coupling LS1 and LS2 in a VND strategy, the resulting algorithm obtains the best outcomes.

The next experiment consists in determining the influence of the parameter k_{max} on the performance of GVNS. In particular, we consider $k_{max} = \{3, 4, 5, 6, 7\}$. In Table 5, we show the associated results. As expected, the higher the value of k_{max}, the better the results. In this case, the Friedman test again highlights the statistically significant differences among the tested algorithms (p-value <0.001), becoming $k_{max} = 7$ the first algorithm in the ranking (2.39), closely followed by $k_{max} = 6$ (2.61) and $k_{max} = 5$ (2.79). The remaining values (i.e., $k_{max} = 4$ and $k_{max} = 3$) have a lower performance with an associated ranking values of 3.43 and 3.79, respectively. It is worth mentioning that the computing time increases with the value of k_{max}, so we set $k_{max} = 5$ as a compromise between computing time and quality.

According to Crainic and Michel (2003), the classical performance measure for parallel algorithms (i.e., speedup described by Barr and Hickman, 1993) is not adequate to evaluate the performance of parallel metaheuristics. This is due to the fact that asynchronous interactions between threads generally induce significant differences in search behavior, not only for the global parallel method, but also for each process participating in cooperation. Therefore, the sequential and parallel methods may then be viewed as different metaheuristics, requiring a redefinition of speedup and other performance measures. This situation is further aggravated by the randomness embedded in the VNS methods considered in this paper. However, it is important to note that a parallelization strategy should accelerate the search or, alternatively, produce better results than the sequential method. Consequently, we compare the quality of the solutions obtained by the sequential and parallel VNS methods to evaluate the quality of the parallel design strategy. In particular, the next experiment compares the performance of the GVNS (Section 4) and its parallel redesign following the RSVNS methodology (Section 5). We conduct this experiment on a computer able to execute up to eight processes (threads) simultaneously. Therefore, we consider one thread (i.e., sequential version), and two, four, and eight threads. In Table 6 we report the associated results, where the parameter in parenthesis indicates the number of threads. Notice that, in this table, RSVNS(1) is equivalent to GVNS.

In Table 6, we show that all parallel versions outperform the sequential VNS method, RSVNS(1), according to the statistics in the table. In this experiment, the p-value associated to the Friedman test is 0.026, which indicates that there are statistically significant differences among all the variants, emerging RSVNS(8) as the best method in the experiment. These results can be partially explained by the fact that the proposed parallel methods explore larger portions of the search space. We also

Table 6
Comparison of different configurations of RSVNS when varying the number of threads

No. of threads	Time (seconds)	Dev. (%)	#Best
RSVNS(1)	43.94	1.73	6
RSVNS(2)	40.01	1.55	6
RSVNS(4)	51.78	0.99	6
RSVNS(8)	78.17	0.36	12

observe that the best outcomes are obtained with eight threads. It presents an average deviation of 0.36% and 12 best-found solutions (out of 14), which compares favorably with the sequential version (1.73% and 6). Computing times of the sequential version and parallel versions with two and four threads are similar (about 50 seconds). This is mainly because the objective of RSVNS is not the reduction of the computing time, but the exploration of a larger portion of the search space. However, the parallel version with eight threads has a considerably larger computing time. This deviation is mainly due to one instance, where the RSVNS(8) uses more than 400 seconds while the computing time of the other three variants is about 80 seconds. However, this extra time drives to a much better local optimum. We therefore select the parallel variant with eight threads as the best procedure for our final experimentation.

6.2. Final experimentation

Once we have identified the best parameters and strategies among our proposed variants, the final experiment is intended to compare the performance of our best proposal with the best previous approaches in the state of the art. In particular, we have selected C1 as a constructive procedure, VND as improvement strategy, and RSVNS with eight threads as the best RSVNS variant, to compose our best algorithm. For the sake of simplicity, our method is named as RSVNS. This approach is then compared to the heuristic algorithm described in Gademann et al. (2001), denoted in the following as Heur-Gademann et al., and with the exact procedure also described in the same paper, denoted as B&B-Gademann et al.

For this final comparison, we considered the full set of 432 instances. In order to have a fair comparison, the heuristic methods are executed for a similar computing time. On the other hand, the exact method has been executed for a maximum computing time of 600 seconds, as suggested by the authors. Results in Table 7 are split into two main groups, since we have considered two routing strategies. Specifically, we test the performance of the three compared methods (Heur-Gademann et al., B&B-Gademann et al., and RSVNS) by joining them with a heuristic and exact routing strategies (combined and Ratliff and Rosenthal methods, respectively). Note that we do not report the optimal value when using the combined strategy, since it does not guarantee the optimal route.

The first conclusion that we can extract by analyzing these results is that the new proposal clearly outperforms the previous methods in the state of the art when considering both routing strategies. In particular, considering the whole set of instances (432), RSVNS obtains the lowest average deviation and the highest number of best solutions found in the same (or even lower) computing time. Specifically, RSVNS achieves an average deviation of 0.17% and 0.14% when using

Table 7
Comparison of our best proposal with the algorithms in the state of the art, considering the combined and Ratliff and Rosenthal routing strategies

Combined routing strategy					
Instances	Algorithm	Time (seconds)	Dev. (%)	#Best	#Opt
AAMD (240)	Heur-Gademann et al.	113.68	1.27	80	–
	B&B-Gademann et al.	473.31	1.17	95	–
	RSVNS	134.78	0.18	211	–
HW (192)	Heur-Gademann et al.	0.95	1.38	49	–
	B&B-Gademann et al.	509.09	0.73	77	–
	RSVNS	2.70	0.15	163	–
Total (432)	Heur-Gademann et al.	64.07	1.13	129	–
	B&B-Gademann et al.	489.21	0.98	172	–
	RSVNS	76.08	0.17	374	–
Ratliff and Rosenthal routing strategy					
Instances	Algorithm	Time (seconds)	Dev. (%)	#Best	#Opt
AAMD (240)	Heur-Gademann et al.	198.03	1.34	69	38
	B&B-Gademann et al.	479.42	1.27	83	51
	RSVNS	227.53	0.16	205	39
HW (192)	Heur-Gademann et al.	9.18	1.15	32	10
	B&B-Gademann et al.	512.31	0.97	58	33
	RSVNS	11.48	0.11	164	11
Total (432)	Heur-Gademann et al.	114.10	1.26	101	48
	B&B-Gademann et al.	494.04	1.13	141	84
	RSVNS	131.50	0.14	369	50

the combined and Ratliff and Rosenthal strategies, respectively. The deviation of the second best method (B&B) is, on average, 1% higher than our proposal in both routing strategies, requiring more than four times the computing time of RSVNS. Regarding the number of best solutions found, we can observe the quality of our proposal, which obtains 374 (combined) and 369 (Ratliff and Rosenthal) of 432 best solutions found, when the B&B is able to obtain only 172 and 141, respectively.

When considering the number of optima, we can only make reference to the results obtained by B&B-Gademann et al. when paired with Ratliff and Rosenthal routing strategy. As expected, the exact procedure finds the largest number of optima (84 out of 432) and, our approach, is able to find 50 of them. Note that since our method finds the largest number of best solutions (369 out of 432), some of them could also be optimal but we cannot certify it because the exact procedure was not able to solve them in allowed computing time.

We confirmed these results by conducting the Friedman statistical test, which ranks the algorithm according to their quality. The ranks obtained when considering the combined strategy are 1.46 (RSVNS), 2.17 (B&B-Gademann et al.), and 2.37 (Heur-Gademann et al.), resulting in a p-value

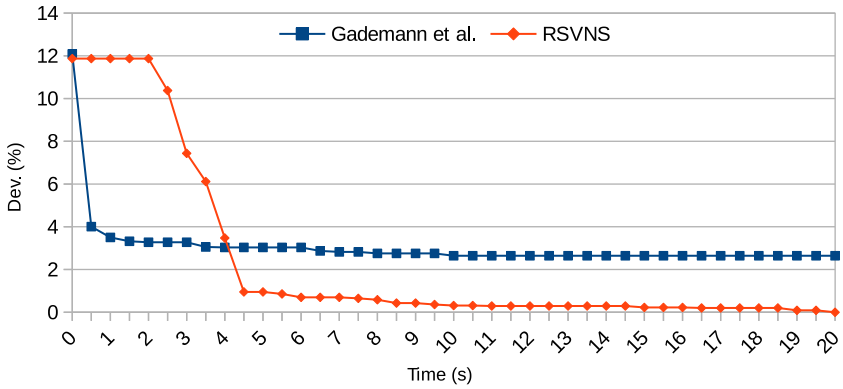

Fig. 9. Search profile of the average deviation, for a 20-second run on eight large instances.

lower than 0.001, which indicates that there are statistically significant differences among the compared methods. Similarly, when considering the Ratliff and Rosenthal strategy, the ranks obtained are 1.42 (RSVNS), 2.18 (B&B-Gademann et al.), and 2.40 (Heur-Gademann et al.), again with a p-value lower than 0.001. These results support the superiority of our proposal, independent of the routing strategy used.

We have additionally conducted a statistical test for pairwise comparisons. In particular, we compared our method (RSVNS) and the best previous algorithm in the state of the art (B&B-Gademann et al.) with the well-known Wilcoxon test. The resulting p-value lower than 0.001 (using both routing strategies) indicates that the values compared come from different populations (using the typical significance level of $\alpha = 0.05$ as the threshold between rejecting or not rejecting the null hypothesis).

Finally, we illustrate the behavior of the two heuristic methods over the time. In Fig. 9, we show the average deviation of the methods with respect to the best-known values. In particular, we report this average every 0.5 seconds over a set of eight representative instances. Again, we left the state-of-the-art method to finish its execution and then adjusted our time to the time needed by that method. As we can observe in the figure, the method by Gademann et al. (2001) produces good-quality results from the very beginning of the search. However, its performance falls drastically after three seconds of execution. On the other hand, our algorithm needs a little more time to produce better results than the method by Gademann et al. (2001), but after four seconds of execution, it consistently produces the best values until the end of the experiment.

7. Conclusions

This paper presented a parallel VNS algorithm for solving the min–max OBP. First, we designed a sequential GVNS. Then, we adapted the previous GVNS to a parallel version based on the RSVNS methodology. To find a good configuration for our best heuristic, we have introduced two constructive procedures based on different greedy strategies, called C1 and C2. We also proposed two local search strategies, LS1 and LS2, based on swap moves. It is important to highlight that

both local search procedures incorporate a new definition of the improvement move, which allows them to deal with flat landscapes. As opposed to the standard way of applying GVNS, where the starting solution is random, the sampled greedy constructed solution is used.

Additionally, we introduced a novel scheme for calculating the objective function, based on a matrix of retrieving times, which substantially reduced the computing time (by a factor of 2) when compared to the straightforward implementation. This new approach opens a new avenue of research with high interest for this field, since many practical applications could make use of the advantages of parallel designs.

The extensive experimental comparison performed showed that the parallel version of the algorithm, RSVNS, clearly outperformed the best previous approaches by Gademann et al. (2001), in the state of the art. These results were confirmed by nonparametric statistical tests, emerging RSVNS as the new state-of-the-art algorithm for the min–max OBP.

Finally, given the positive results obtained with the parallel implementation of VNS, we note the possibility of applying the presented strategies to other variants of this problem, which are already in the literature. Additionally, our parallel approach is not unique. The study of other parallel strategies, such as RPVNS or CNVNS open the door to a wide variety of possibilities that invite closer examination and may give an interesting basis for future research.

Acknowledgment

This research has been partially supported by the Spanish "Ministerio de Economía y Competitividad," and "Comunidad de Madrid" with grants refs. TIN2015-65460-C2-2-P and S2013/ICE-2894, respectively.

References

Alba, E., 2005. *Parallel Metaheuristics: A New Class of Algorithms*, Vol. 47. John Wiley & Sons, Hoboken, NJ.

Albareda-Sambola, M., Alonso-Ayuso, A., Molina, E., De Blas, C., 2009. Variable neighborhood search for order batching in a warehouse. *Asia-Pacific Journal of Operational Research* 26, 655–683.

Azadnia, A.H., Shahrooz, T., Pezhman, G., Saman, M., Zameri, M., Wong, K.Y., 2013. Order batching in warehouses by minimizing total tardiness: a hybrid approach of weighted association rule mining and genetic algorithms. *The Scientific World Journal* 2013, Article ID 246578.

Barr, R., Hickman, B., 1993. Reporting computational experiments with parallel algorithms: issues, measures, and experts opinions. *ORSA Journal on Computing* 5, 2–18.

Caporossi, G., Cvetkovic, D., Gutman, I., Hansen, P., 1999. Variable neighborhood search for extremal graphs. 2. Finding graphs with extremal energy. *Journal of Chemical Information and Computer Sciences* 39, 984–996.

Chen, F., Wang, H., Xie, Y., Qi, C., 2016. An ACO-based online routing method for multiple order pickers with congestion consideration in warehouse. *Journal of Intelligent Manufacturing* 27, 2, 389–408.

Chen, T.L., Cheng, C.Y., Chen, Y.Y., Chan, L.K., 2015. An efficient hybrid algorithm for integrated order batching, sequencing and routing problem. *International Journal of Production Economics* 159, 158–167.

Clarke, G., Wright, J., 1964. Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research* 12, 568–581.

Cook, S., 2012. *CUDA Programming: A Developer's Guide to Parallel Computing With GPUs (Applications of GPU Computing)* (1st edn). Morgan Kaufmann Publishers, San Francisco, CA.

Crainic, T., Gendreau, M., Hansen, P., Mladenović, N., 2004. Cooperative parallel variable neighborhood search for the *p*-median. *Journal of Heuristics* 10, 293–314.

Crainic, T., Michel, T., 2003. Parallel strategies for meta-heuristics. In Glover, F., Kochenberger, G. (eds) *Handbook of Metaheuristics*, Vol. 57, *International Series in Operations Research & Management Science*. Springer, Boston, MA, pp. 475–513.

De Koster, R., Roodbergen, K., van Voorden, E., 1999a. Reduction of walking time in the distribution center of De Bijenkorf. In Speranza, M.G., Stähly, P. (eds) *New Trends in Distribution Logistics*. Springer, New York, pp. 215–234.

De Koster, R., Van der Poort, E., Wolters, M., 1999b. Efficient orderbatching methods in warehouses. *International Journal of Production Research* 37, 1479–1504.

Drury, J., 1988. *Towards More Efficient Order Picking*. IMM Monograph No. 1, The Institute of Materials Management, Cranfield.

Duarte, A., Escudero, L.F., Mladenović, N., Pantrigo, J., Sánchez-Oro, J., 2012. Variable neighborhood search for the vertex separation problem. *Computers and Operations Research* 39, 3247–3255.

Elsayed, E., Lee, M.K., Kim, S., Scherer, E., 1993. Sequencing and batching procedures for minimizing earliness and tardiness penalty of order retrievals. *International Journal of Production Research* 31, 727–738.

Gademann, N., Van Den Berg, J., Van Der Hoff, H., 2001. An order batching algorithm for wave picking in a parallel-aisle warehouse. *IIE Transactions* 33, 385–398.

Gademann, N., Velde, S., 2005. Order batching to minimize total travel time in a parallel-aisle warehouse. *IIE Transactions* 37, 63–75.

Gao, G., Sato, M., Ayguadé, E., 2008. Special issue on parallel programming with OpenMP. *International Journal of Parallel Programming* 36, 287–288.

García-López, F., Melián-Batista, B., Moreno-Pérez, J., Moreno-Vega, J., 2002. The parallel variable neighborhood search for the p-median problem. *Journal of Heuristics* 8, 375–388.

Gibson, D., Sharp, G., 1992. Order batching procedures. *European Journal of Operational Research* 58, 57–67.

Glover, F., 1992. New ejection chain and alternating path methods for traveling salesman problems. *Computer Science and Operations Research* 449, 491–507.

Goetschalckx, M., Ratliff, H.D., 1988. Order picking in an aisle. *IIE Transactions* 20, 53–62.

Henn, S., 2012. Algorithms for on-line order batching in an order picking warehouse. *Computers & Operations Research* 39, 2549–2563.

Henn, S., 2015. Order batching and sequencing for the minimization of the total tardiness in picker-to-part warehouses. *Flexible Services and Manufacturing Journal* 27, 1, 86–114.

Henn, S., Koch, S., Doerner, K., Strauss, C., Wäscher, G., 2010. Metaheuristics for the order batching problem in manual order picking systems. *BuR Business Research Journal* 3, 1, 82–105.

Henn, S., Schmid, V., 2013. Metaheuristics for order batching and sequencing in manual order picking systems. *Computers & Industrial Engineering* 66, 338–351.

Henn, S., Wäscher, G., 2012. Tabu search heuristics for the order batching problem in manual order picking systems. *European Journal of Operational Research* 222, 484–494.

Hoos, H., Stützle, T., 2004. *Stochastic Local Search: Foundations & Applications*. Morgan Kaufmann Publishers, San Francisco, CA.

Hsu, C.M., Chen, K.Y., Chen, M.C., 2005. Batching orders in warehouses by minimizing travel distance with genetic algorithms. *Computers in Industry* 56, 169–178.

Lozano, M., Duarte, A., Gortázar, F., R., M., 2012. Variable neighborhood search with ejection chains for the antibandwidth problem. *Journal of Heuristics* 18, 919–938.

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

Oaks, S., Wong, H., 2004. *Java Threads*. O'Reilly Media, Sebastopol, CA.

Öncan, T., 2015. MILP formulations and an iterated local search algorithm with tabu thresholding for the order batching problem. *European Journal of Operational Research* 243, 1, 142–155.

Ratliff, H.D., Rosenthal, A.S., 1983. Order-picking in a rectangular warehouse: a solvable case of the traveling salesman problem. *Operations Research* 31, 507–521.

Roodbergen, K., De Koster, R., 2001. Routing methods for warehouses with multiple cross aisles. *International Journal of Production Research* 39, 1865–1883.

Rubrico, J., Toshimitu, H., Hirofumi, T., Jun, O., 2011. Online rescheduling of multiple picking agents for warehouse management. *Robotics and Computer-Integrated Manufacturing* 27, 62–71.

Sánchez-Oro, J., Mladenović, N., Duarte, A., 2015. General variable neighborhood search for computing graph separators. *Optimization Letters*, doi:10.1007/s11590-014-0793-z.

Sánchez-Oro, J., Pantrigo, J.J., Duarte, A., 2014. Combining intensification and diversification strategies in VNS. An application to the vertex separation problem. *Computers & Operations Research* 52, Part B, 209–219.

Talbi, E.G., 2009. *Metaheuristics: From Design to Implementation*. John Wiley & Sons, Hoboken, NJ.