Intl. Trans. in Op. Res. xx (2017) 1–23
DOI: xx.xxxx/itor.xxxxx

A Parallel Variable Neighborhood Search Approach for the Obnoxious p-Median Problem

Alberto Herrán[a], José M. Colmenar[a], Rafael Martí[b] and Abraham Duarte[a,*]

[a] *Dept. Computer Sciences, Universidad Rey Juan Carlos, Madrid, Spain*
[b] *Departamento de Estadística e Investigación Operativa, Universidad de Valencia, Valencia, Spain.*
E-mail: alberto.herran@urjc.es [A. Herrán]; josemanuel.colmenar@urjc.es [J.M. Colmenar];
rafael.marti@uv.es [R. Martı]; abraham.duarte@urjc.es [A. Duarte]

Received DD MMMM YYYY; received in revised form DD MMMM YYYY; accepted DD MMMM YYYY

Abstract

The Obnoxious p-Median Problem consists of selecting p locations, considered facilities, in a way that the sum of the distances from each non-facility location, called customers, to its nearest facility is maximized. This is an $\mathcal{NP}$-hard problem that can be formulated as an integer linear program. In this paper, we propose the application of a Variable Neighborhood Search (VNS) method to effectively tackle this problem. First, we develop new and fast local search procedures to be integrated into the Basic VNS methodology. Then, some parameters of the algorithm are tuned in order to improve its performance. The best VNS variant is parallelized and compared with the best previous methods, namely Branch and Cut, Tabu Search and GRASP over a wide set of instances. Experimental results show that the proposed VNS outperforms previous methods in the state of the art. This fact is finally confirmed by conducting non-parametric statistical tests.

Keywords: Obnoxious location, Metaheuristics, VNS, Parallel algorithms.

1. Introduction

The Obnoxious p-Median problem, OpM, (Church and Garfinkel, 1978; Erkut and Neuman, 1989), belongs to the category of facility location problems (Francis and White, 1974). More precisely, it is derived from the p-median problems (Hakimi, 1964) which, for fixed values of p, they can be solved in polynomial time. On the other hand, they become strongly $\mathcal{NP}$-hard for variable values of p (Current et al., 2004).

In OpM, p identifies those facilities that are actually opened, whereas the rest ones are considered as unopened facilities. In addition, it considers the open facilities to be obnoxious. That is, they correspond

* Author to whom all correspondence should be addressed (e-mail: abraham.duarte@urjc.es).

to those real-world facilities that are usually noisy, dangerous, or disgusting for humans, like waste disposal facilities or airports, for instance. Therefore, instead of trying to reduce the distance between open facilities and customers, as it happens in classical location problems, the objective is to maximize the distance between the facilities and their closer customer.

More formally, we can define the problem as follows. Let I be a set of clients, J a set of facilities, and d_{ij} the distance between the client $i \in I$ and the facility $j \in J$. The OpM problem consists in finding a subset S of the set of facilities with $|S| = p$, $S \subseteq J$ and $p < |J|$, such that the sum of the minimum distance between each client and the set of facilities is maximized. In mathematical terms, if we call f to the objective function, the problem is defined as:

$$\max \ f(S) = \sum_{i \in I} \min\{d_{ij} : j \in S\}$$

subject to

$$S \subseteq J \ / \ |S| = p$$

The OpM problem was introduced in Belotti et al. (2007) as belonging to the same family of the p-dispersion and the p-median optimization problems (Shier, 1977; Kuby, 1988). As stated in Belotti et al. (2007), the OpM problem belongs to the class of maxi-sum (or *p*-antimedian) problems. In Ting (1988) it is proven that this problem is polynomially solvable on networks with a $O(n)$ algorithm in the case of $p = 1$ (1-maxian problem). However, in Tamir (1991) it is proven that the general and discrete cases of both maxi-sum and maxi-min problems are $\mathcal{NP}$-hard. Besides, for the simplest cases where $p = 1$ and $p = 2$, they present a polynomial-time algorithm.

Note that the practical need to solve the OpM problem may involve medium and large size instances, which motivates the use of heuristics. The exact method by Belotti et al. (2007) is able to solve medium size instances (up to $p = 50$). For larger instances, their method provides a gap w.r.t. the best upper bound. Therefore, if we need to know a good solution for instances with p larger than 100, it is useful to run a heuristic method to complement the information provided by the exact method. For example, in the well-known example of nuclear plant allocation, we find around $p = 150$ plants in Europe, which would require the use of both methodologies, exact and heuristic, to determine a high-quality solution for such an important problem. The elaboration of rational policies in the European context motivates the need of efficient methods to this problem

An extensive literature review is given in relation to those kind of problems. In addition, a mathematical model, a linear programming formulation, and some valid inequalities are provided. These theoretical results are then used in Belotti et al. (2007) to implement a Branch & Cut (BC) method. Besides, a tabu-search heuristic (XTS) is also introduced. In order to reduce the executing time, XTS is coupled with a BC implementation. As it is shown in the associated computational experience, this hybrid algorithm obtains good results in terms of quality. The study is performed over a set of instances derived from the p-median problem literature.

A GRASP method has been recently presented in (Colmenar et al., 2016). The authors present two different constructive methods and two local search algorithms. Additionally, they propose an efficient strategy to update the objective function value that considerably reduces the running time of GRASP.

Finally, the authors describe a filtering strategy intended to discard low-quality solutions and to selectively apply the local search only to promising solutions. The experimental results show that the GRASP method is able to outperform BC and XTS in both, short and long time horizons. The comparison is conducted over a set of larger instances than those used in Belotti et al. (2007). Notice that these instances are also derived from the p-median literature.

In this paper, we propose a Variable Neighborhood Search (VNS) method to tackle the OpM problem. Specifically, we propose new and fast local search procedures that explore a reduced neighborhood. In addition, we propose a parallel VNS method, where we investigate different strategies of exchanging solutions among parallel executions. Finally, we compare the VNS proposal with the current state-of-the-art methods over the set of previously reported instances. Our results show that the VNS approach obtains the best solution in 137 out of the 144 instances in this benchmark. In addition, the VNS algorithm is faster that the competitors in all the instances where it obtains the best solution.

The rest of the paper is organized as follows. Section 2 describes the VNS proposal. Section 3 shows how to improve the efficiency of the basic VNS described in Section 2 by using Multiple Markov Chains. Section 4 shows the details of the parallel implementation of the algorithm described in Section 3. Section 5 details the experimentation and the analysis of the results. Section 6 draws the conclusions and identifies future lines of research. Finally, in the Appendix, a breakdown of the results is presented, detailing the best cost and execution time obtained for each one of the instances under study.

2. Basic Variable Neighborhood Search

The Variable Neighborhood Search (VNS) metaheuristic is a general purpose optimization methodology which considers changes of neighborhood in both, the descent phase (to find a local optimum), and the perturbation phase (to get out of the corresponding basin of attraction) (Hansen and Mladenovic, 2001). The original metaheuristic has evolved in different ways, giving rise to many extensions: Variable Neighborhood Descent (VND), Reduced VNS (RVNS), Basic VNS (BVNS), Skewed VNS (SVNS), General VNS (GVNS), Variable Neighborhood Decomposition Search (VNDS) or Reactive VNS, among others. See (Hansen et al., 2008; 2010) for a recent thorough review. VNS has been successfully applied to a wide variety of optimization problems such the clustered vehicle problem (Defryn and Sorensen, 2017), the maximum min-sum dispersion problem (Amirgaliyeva et al., 2016) or the job scheduling problems with distinct time windows and penalties (Rosa et al., 2017), just to cite several recent works.

In this paper, we consider the Basic VNS variant, as shown in Algorithm 1. It starts by setting the maximum value of the parameter that controls the perturbation strength of the shaking mechanism k_{max}. It is worth mentioning that this stage is tied to a portion β of the number of opened facilities p (step 1). Then, the algorithm builds a solution through a constructive method (step 2). We apply the procedure described in (Colmenar et al., 2016), named as C2. It randomly selects a fraction of the facilities to be included in the solution. Then, it evaluates all of them, computing the greedy function that measures the contribution to the quality of the solution, and selects the best one. The procedure ends when the solution contains exactly p facilities.

Once a feasible solution is constructed, the algorithm initializes the search within the first neighborhood, by setting k to 1 (step 4). The algorithm then iterates until k reaches its maximum value (steps 5 to 17). At each iteration, the current solution is perturbed by the shake procedure in order to obtain a new

solution (step 7). The shaking mechanism of the VNS procedure is intended to diversify the search. In particular, it receives as input argument a feasible solution S, and uses randomly sorted working copies of the set of facilities J (i.e. the set of opened and unopened facilities). Then, it simply performs k pairwise exchanges between facilities in both sets.

During preliminary experimentation, we observed that the shake procedure coupled with the local search method yielded to the same solutions more than once. In order to avoid the application of a time consuming procedure (i.e, the local search method) to a solution more than once, we developed a filtering method based on a hash function. Specifically, a hash value is computed and stored for each solution generated during the search. The hash value of a solution that is candidate for improvement is checked against the database H of previously generated hash values. If a match is found then the solution under consideration is not subjected to the improvement method. If not, the hash value of the solution is added to H and the algorithm tries to improve the solution by using the local search (steps 10 and 11). Then, it checks whether the given solution S'' improves the best so far S, restoring k to its initial value (steps 13 to 14); otherwise, the value of k is increased by 1 to perform a neighborhood change (step 16). Finally, steps 4 to 17 are repeated until the computation time exceeds a maximum value of t_{max} as termination criteria.

Algorithm 1: VARIABLE NEIGHBORHOOD SEARCH (VNS)

```
1  k_max ← β · p
2  S ← Constructive()
3  while t < t_max do
4      k ← 1
5      repeat
6          do
7              S' ← Shake(S, k)
8              h ← Hash(S')
9          while isInHashSet(h, H)
10         H ← H ∪ {h}
11         S'' ← LocalSearch(S')
12         if f(S'') < f(S) then
13             S ← S''
14             k ← 1
15         else
16             k ← k + 1
17     until k = k_max
18 return S
```

The local search stage in VNS tries to improve the incumbent solution by exploring the neighborhoods generated by some move operator. So far, all local search methods designed for the OpM have used an exchange operator, that basically interchanges a facility j (selected from the set of opened facilities S) with a different one j' (selected from the set of unopened facilities $J \setminus S$). This move produces a new

feasible solution $S' = S \setminus \{j\} \cup \{j'\}$. It is possible to efficiently evaluate such a move without recomputing the objective function from scratch, following the so-called incremental evaluation. Specifically, as described in Colmenar et al. (2016), we make use of a list for each client that stores the current best link to the closest facility. More precisely, these distances are separated in two independent data structures. The first one contains the distances from all clients to the opened facilities (sorted in ascending order). Similarly, the distances from clients to unopened facilities are also stored in another sorted data structure. Considering this implementation, an exchange move value can be incrementally computed in $O(|I|\log(|J|))$.

In this paper, we consider a reduction of the size of this standard neighborhood to further reduce the computing time of the associated local search procedure. More precisely, given a solution S, its corresponding neighborhood (i.e., those solutions that are reached by using the exchange move) has a size of $p\times(|J|-p)$. This number is obtained by considering the interchanges between the p open facilities with the $|J| - p$ unopened facilities. Given this size, it could be computationally expensive to identify the best move at each iteration of local search. To overcome this obstacle, we split the corresponding compound move into two simple operations. Specifically, we first evaluate the effect of removing an opened facility j, selecting the one that produces the minimum decrement of the objective function. We then evaluate the effect of including an unopened facility j', selecting the one that produces the largest increment in the objective function. The associate size of this neighborhood is then $p + (|J| - p) = |J|$, which is smaller than the classical one.

Algorithm 2 shows the pseudo-code for the first proposed reduced local search, denoted as RLS1. After setting the improvement flag to false (step 3), it iterates over the open facilities in order to find a facility j_{drop} which produces the smallest decrement in the objective function of the resulting unfeasible solution (steps 5 to 9). Once the selected facility is dropped (step 10), the next step is to bring the solution back to the feasible region by adding the unopened facility j_{add}. RLS1, selects the one that produces the largest increment of the objective function of the resulting feasible solution (steps 12 to 16). Functions $drop()$ and $add()$ evaluate the variation of the objective function (denoted as $\triangle f$) of a solution when dropping or adding a facility. These functions are evaluated in an incremental way. Then, RLS1 tests whether S' improves upon S, in which case the incumbent solution is updated and the improvement flag is set to true (steps 19 and 20). The loop (steps 2 to 20) is repeated until no improvement is meet, and then, the improved solution is returned (step 21).

The second variant of reduced local search is called RLS2, and it works backwards. That is, it firstly adds the unopened facility which produces the largest increment in the objective function, making the solution unfeasible. Then, the feasibility is restored by removing the open facility which produces the smallest decrement of the objective function. We do not provide the pseudo-code for RLS2, since it is similar to RLS1, with the minor difference of exchanging the order of simple operations. Notice that, despite the fact that the local search methods seem to be similar, they are not, because the move value variation when adding of dropping is different, since it is calculated based on an unfeasible solution with different size in RLS1 and RLS2.

Algorithm 2: REDUCED LOCAL SEARCH 1 (RLS1)

```
1  improve ← true
2  while improve do
3      improve ← false
4      △f_drop ← ∞
5      foreach j ∈ S do
6          △f ← drop(j, S)
7          if △f < △f_drop then
8              j_drop ← j
9              △f_drop ← △f
10     S' ← S \ {j_drop}
11     △f_add ← 0
12     foreach j ∈ J \ S do
13         △f ← add(j, S')
14         if △f > △f_add then
15             j_add ← j
16             △f_add ← △f
17     S' ← S' ∪ {j_add}
18     if f(S') > f(S) then
19         S ← S'
20         improve ← true
21 return S
```

3. Multi-start Variable Neighborhood Search

As described in (Martí et al., 2017), the re-start mechanism of multi-start methods can be superimposed on many different search methods. Once a new solution has been generated, a variety of options can be used to improve it, ranging from a simple greedy routine to a complex metaheuristic. Multi-start methods basically provide an efficient framework to achieve global diversification in the search process.

We can consider steps 6 to 16 of Algorithm 1 as an iteration r of the VNS method that, from an solution S, it returns a new solution S''. In this way, we can view the entire VNS as an algorithm generating a sequence S_0, S_1,..., S_r of solutions that are improving its value. This sequence is generated by iteratively moving to a neighbor solution, which is accepted under the criteria shown in step 12. Since the choice of S_{r+1} depends only on the current solution S_r but not on the previously visited ones, the search path of such an algorithm follows a first order Markov Chain (Norris, 1998). These procedures used to be referenced as Sequential Single Markov Chain algorithms (SSMC). In these methods, the probability of not getting an optimal solution after r iterations is characterized by Eq.(1), where S_{min} is a set of optimal solution points and C and θ are positive constant values associated to a given objective function and neighborhood generation method respectively. Thus when $r \to \infty$, the solution converges to one of the optimal points in S_{min} with a probability of 1.

$$P(S_r \notin S_{min}) \approx \left(\frac{C}{r}\right)^{\theta} \tag{1}$$

Although many approaches in the literature follow this schema, using a single Markov Chain could be inefficient from an efficient point of view (Defersha, 2008). To solve this problem, a Sequential Multiple Markov Chain algorithm (SMMC) performs V independent executions of the SSMC algorithm, using the same search space and neighborhood exploration method. Each one of these independent versions is stopped after r iterations to provide V terminal solutions, out of which, the best one is chosen as the final solution. Now, the probability of not getting an optimal solution after r iterations shown in Eq.(1) becomes the one shown in Eq.(2). Thus, for $0 < C/r < 1$, this probability decreases exponentially as V increases (Azencott, 1992), while the CPU-time increases only linearly with V. Hence, given a CPU-time t_{max}, it is more efficient to run r/V iterations in a SMMC than run r iterations in a SSMC. Following this schema, we propose a multi-start VNS (instead of the traditional sequential implementation) based on the theoretical results found in the context of SMMC.

$$P(S_r \notin S_{min}) = \prod_{v} P(S_{r,v} \notin S_{min}) \approx \left(\frac{C}{r}\right)^{\theta V} < \left(\frac{C}{r}\right)^{\theta} \tag{2}$$

An additional advantage of this strategy is that iterations of single VNS methods are completely independent. Therefore, this kind of algorithms are suitable to be implemented in parallel. In the context of metaheuristics, distributed algorithms are those in which V independent algorithms are partitioned into G groups of V/G single algorithms each, communicating them at every R iterations in order to further improve the solution quality. This kind of algorithms have been extensively studied by the Genetic Algorithms community (Alba and Troya, 1999). Alba and Troya (2000) highlights that the best performance of using this strategy is attributed to the following characteristics: (1) their decentralized search which allows more specialization; (2) the larger diversity levels, since many search regions are sought at the same time; and (3) the exploitation inside each group by refining its best partial solution found.

The communication among groups is based on a migration operator, where the best solutions of one group are sent to another according to different interaction topologies. The most commonly used are: *Fully Connected* (FC), *Ring* (RG), and *Master Slave* (MS). We illustrate these topologies in Figure 1 for G=4 and V=20. See (Nowostawski and Poli, 1999) for further details. In the FC topology, the best solution found in each group is sent to all others. The RG scheme is similar to FC but only sending the best solution found in each group g to its neighbors ($g-1$ and $g+1$). In the MS scheme, the best solution so far (found among all groups) determines the master group, and then, this solution is sent to all of the remaining ones, the slaves. In all the topologies, each time a group receives a solution, it is only accepted if it improves the worst solution of all the V/G single algorithms running within it. We call our VNS version that follows this schema Distributed Multi-start VNS.

As discussed above, multi-start versions of VNS may result in an exponential reduction of the error probability with a linear increasing of computational time as the number of independent V single VNS runs increases. As a direct consequence, convergence behavior and the robustness of the algorithm are improved by using multiple short runs in a VNS instead of one long run of a single VNS executed during

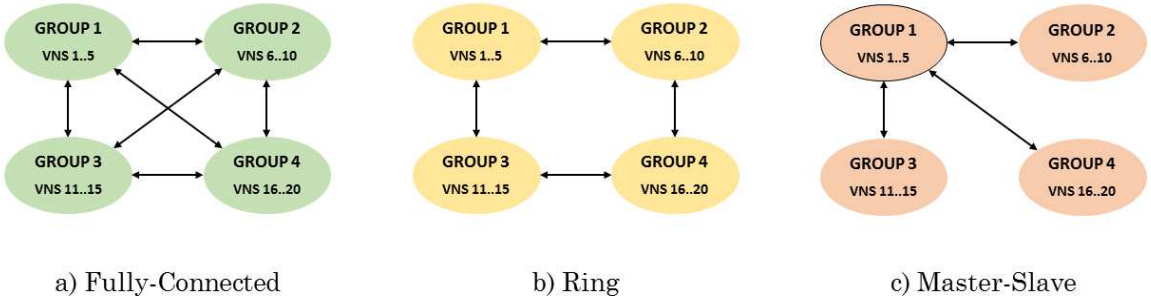

Fig. 1. Different communication topologies: (a) Fully-Connected, (b) Ring, and (c) Master-Slave.

the same total CPU-time. Indeed, considering that VNS runs are independent, a promising technique to achieve this exponential reduction of the error probability with less or no increment of computational time is to use parallel computing.

4. Parallel Variable Neighborhood Search

Modern commodity-computers are now able to execute different programs simultaneously, since they have several processors (cores). Computer scientists have been using this capability to increase the performance of their algorithms. This fact might be particularly effective in the case of metaheuristics. However, this task is far for being trivial since, in general, it implies algorithmic redesign to be adapted to the specific architecture.

The first attempt to parallelize VNS was presented in García-López et al. (2002). In particular, three different approaches were described: 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. The second approach, RPVNS, explores a wider portion of the solution space by using a multi-start strategy. The last variant, RSVNS, follows a classical master-slave scheme, where the master executes the VNS, and each slave executes the shake and local search methods.

The authors in 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.

In Duarte et al. (2016), three different strategies to parallelize VNS were described. The first one is oriented to parallelize the whole VNS method. The second one parallelizes the shake and the local search procedures. Finally, the third one explores in parallel the set of predefined neighborhoods. The

authors illustrate the performance of these strategies by considering a previous state-of-the-art algorithm designated for the cutwidth minimization problem (Pardo et al., 2013).

A parallel VNS strategy for the min-max order batching problem was also presented in Menéndez et al. (2017). This optimization problem involves two main actions: on the one hand, to group orders into feasible batches and, on the other hand, to retrieve the items in each order from their location, satisfying some constraints. The objective function consists in minimizing the maximum retrieving time for all the generated batches. These authors analyze all previous parallel strategies concluding that the Replicated Shaking VNS scheme is the most adequate one for the tackled problem.

A different parallel VNS method to solve the dynamic memory allocation problem was proposed in Sánchez-Oro et al. (2015) and Sánchez-Oro et al. (2017). This problem consists of determining the memory state (i.e., location of data structures in memory banks) of a device during the time. The evaluation considers the cost of moving a data structure from a memory bank to another and the cost for executing each instruction of the program. As in the aforementioned work, (Sánchez-Oro et al., 2015), the authors also follow the Replicated-Shaking methodology.

In this paper, we propose a Distributed parallel VNS algorithm (DVNS) which consists of multiple sequential VNS algorithms distributed among G groups concurrently running on the available processors/cores. Algorithm 3 shows a pseudo-code of DVNS following the parallel schema described above. After setting the same k_{max} parameter for all single VNS algorithms (step 1), DVNS constructs an initial solution and sets the initial value of k for each single VNS in each group (steps 2 and 5). Then, the algorithm sets r to 1 (step 6), and repeats the main loop (steps 8 to 13) until the termination criterion is meet. This loop includes the iteration of all the solutions included in each group (step 10), and the communication between these groups at every R iterations (step 12).

The *IterateVNS*() procedure performs steps 6 to 16 of Algorithm 1 for each available solution, while the *Communicate*() procedure shares the best solution found in each group according to some of the migration policies described above (i.e. FC, RG, or MS). Then, the iteration number r is increased by 1 (step 13), testing the termination criterion. Most common finalization criteria include a maximum number of iterations, CPU-time, or finishing the algorithm when all the single VNS of this distributed version reach the maximum value of k. The algorithm ends by returning the best solution found among the G groups (step 15).

Since steps 8 to 10 of the DVNS depicted in Algorithm 3 perform $G \cdot V$ independent iterations of each available solution, they could be executed in parallel if several processors/cores are available. There exist different technologies for the implementation of parallel algorithms, such as *Threads* (Butenhof, 1997), *OpenMP* (Dagum and Menon, 1998), or *CUDA* (NVIDIA Corporation, 2017). In this paper, we use *OpenMP*, which is a set of compiler directives and library routines that can be used to implement parallel algorithms. The algorithm designer then includes these compiler directives within a sequential method in order to inform the compiler which part of the code must be concurrently executed. The main advantage of this technology is the simplicity of its implementation. In other words, transforming a sequential algorithm to the parallel version can be performed by including only compiler directives, without modifying the original sequential code. Therefore, we consider that *OpenMP* is adequate if the algorithm follows a data parallel model. In our case, we can parallelize the Algorithm 3 just by including the correct compiler directives to run step 10 in parallel.

Hence, the parallel version of the DVNS described above has a master process, created by the main algorithm, that follows the *fork-join* model (McCool et al., 2012). This strategy is illustrated in Figure 2.

Algorithm 3: DISTRIBUTED VNS (DVNS)

```
1  k_max ← β · p
2  for g = 1 to G do
3      for v = 1 to V do
4          S_{g,v} ← Constructive()
5          k_{g,v} ← 1
6  r ← 1
7  repeat
8      for g = 1 to G do
9          for v = 1 to V do
10             IterateVNS(S_{g,v}, k_{g,v});
11     if mod(r, R) = 0 then
12         Comunicate(S)
13     r ← r + 1
14 until TerminationCondition()
15 return Best(S)
```

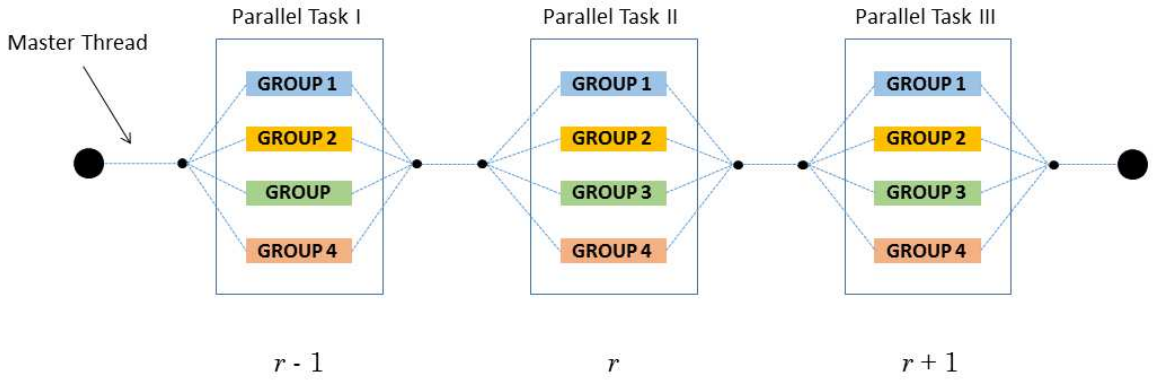

Fig. 2. Fork-Join model followed by the parallel implementation of the DVNS algorithm.

Each time the program enters in the parallel section, it creates a set of processes to run them in parallel across all the available processors (*fork*). Once the parallel section ends, they are synchronized (*join*) and the execution continues in the master thread until it reaches again the next parallel section. The algorithm finally ends when the stopping condition is met, returning the best solution found during the whole search process.

5. Experimental Results

The performance of the algorithm presented in this paper will be illustrated by experimenting with a set of 144 instances, where the number of nodes ranges from 400 to 900. Table 8 in the Appendix reports the main characteristics of each instance, where n indicates the number of nodes, $|I|/|J|$ represents the number of clients/facilities, and p the number of required facilities. These instances were originally introduced in Colmenar et al. (2016), generated with the method described in (Belotti et al., 2007). All these instances are publicly available at www.optsicom.es/opm to facilitate future comparisons.

In order to avoid the over-fitting of our methods, we consider in the preliminary studies a representative subset of 16 instances (i.e., 11%) of the whole set of instances, with different sizes and properties. Notice that the remaining 128 out of 144 instances (89%) are reserved for the final comparison with the state-of-the-art procedures. Table 8 shows in bold font the 16 instances that form the representative subset.

We test the performance of the proposed VNS variants by comparing them with state-of-the-art methods, i.e., XTS and GRASP procedures described in (Belotti et al., 2007) and Colmenar et al. (2016) respectively. All our VNS variants were coded in C++ and the experiments were conducted on an Intel i5 660 processor running at 3.3 GHz with 8 Gb of RAM using GNU/Linux. In order to provide a fair comparison we also executed algorithms XTS (Belotti et al., 2007) and GRASP (Colmenar et al., 2016) in the same computer.

5.1. Preliminary study

In the first preliminary experiment, we analyze the performance of local search methods, RLS1 and RLS2, when they explore the reduced neighborhoods described in Section 2. To this purpose, we run multi-start versions of each local search, starting each one from random feasible solutions. We also include in this comparison previous methods. In particular, the ones presented in Colmenar et al. (2016), denoted as *Local Search Best* (LSB) and *Local Search First* (LSF). Instead of running all the algorithms for a fixed number of restarts, we first run the slower approach (LSB) with 100 restarts. Then, the remaining approaches (LSF, RLS1, and LS2) are executed for the same CPU-time. To test the robustness of the compared methods, we have executed them for 10 independent iterations. Results are shown in Table 1, where the first three columns report the instance identifier, the average running time (in seconds) of each method over each instance (t_{max}, determined by running LSB with 100 restarts) and the best cost (objective function value) found in this experiment for each instance (Best). The following four main columns, show the results of each local search based method. To analyze the effectiveness and efficiency, we report for each method the number of restarts and the best cost (#Exe, Cost) in each instance. Last two rows of this table summarize the results averaged over the subset of 16 test instances (Avg. Values) and the average percentage deviation with respect to the best solution found in this experiment, Dev.(%). As we can see RLS1 and RLS2 are considerable faster (i.e., larger number of #Exe) than LSB and LSF without losing too much quality. Both strategies, based on the exploration of the reduced neighborhood, are even better than LSB approach and slightly worse than LSF. For the sake of brevity we select the local search methods that obtain the best results in terms of quality (LSF) and speed (RLS2).

In the next experiment, we study the performance of the Basic VNS method when using different values of k_{max} and different local search algorithms (LSF and RLS2). In order to adapt the explored

Table 1
Performance of Reduced Local Search (RLS1 and RLS2) against standard Best (LSB) and First (LSF) neighborhood exploration methods over the full set of test instances.

			LSB		**LSF**		**RLS1**		**RLS2**	
Test Instance	t_{max}	**Best**	**#Exe**	**Cost**	**#Exe**	**Cost**	**#Exe**	**Cost**	**#Exe**	**Cost**
pmed17-p25A	7.2	7317	100	7314.3	104	7317.0	480	7314.0	494	7315.8
pmed20-p50A	9.7	5872	100	5858.9	119	5870.2	398	5861.6	402	5866.8
pmed22-p62A	34.0	5995	100	5995.0	159	5995.0	725	5995.0	731	5995.0
pmed28-p75A	36.4	5681	100	5663.4	122	5672.5	471	5667.4	474	5666.6
pmed33-p87A	64.0	5790	100	5773.6	136	5781.5	527	5773.4	537	5776.3
pmed36-p100A	89.7	6461	100	6457.5	119	6461.0	478	6461.0	486	6460.6
pmed39-p112A	200.5	5935	100	5922.7	144	5930.7	752	5925.6	756	5926.2
pmed40-p225A	310.9	4567	100	4533.4	174	4552.5	738	4542.6	729	4540.0
pmed17-p25B	6.2	6905	100	6905.0	112	6905.0	427	6905.0	437	6905.0
pmed20-p50B	10.1	5665	100	5647.8	123	5662.0	420	5643.1	426	5658.1
pmed22-p62B	25.9	6259	100	6259.0	123	6259.0	564	6259.0	572	6259.0
pmed28-p75B	39.0	5642	100	5639.6	123	5642.0	507	5639.6	509	5641.3
pmed33-p87B	88.5	5840	100	5827.7	148	5831.9	704	5831.0	657	5831.0
pmed36-p100B	111.7	6219	100	6210.6	132	6212.8	591	6212.6	587	6212.6
pmed39-p112B	154.8	6198	100	6196.6	136	6197.7	561	6197.4	583	6197.7
pmed40-p225B	309.4	4502	100	4462.1	173	4487.8	717	4485.0	709	4483.8
Avg. Values	93.6	5928.0	100	5916.7	134	**5923.7**	566	5919.6	**568**	5921.0
Dev. (%)				0.191		**0.073**		0.142		0.118

neighborhood to the specific size of the problem, we set $k_{max} = \beta \cdot p$, with β ranging from 0.1 to 0.4. Small values of β promote intensification, while the larger ones favor diversification. As in the previous experiment, we first run the slower VNS version (LSF, $\beta = 0.4$), and then we use its CPU-time as the stopping criterion for the other variants. We show in Table 2 the results of each method averaged over the set of 16 test instances. The average execution time of all of them is 79.8 seconds. Again, last two rows summarize the results of the whole table. We report the average cost, Avg. Cost., the average percentage deviation, Dev.(%), computed with respect to the best solutions found in this experiment, and the averaged success rate, Avg. Succ., calculated as the number of runs (out of 10) in which the algorithm reaches the best solution found in all the experiments for each instance.

Table 2
Performance of VNS-LSF and VNS-RLS2 for different values of β

			VNS-LSF								VNS-RLS2							
			β = 0.1		β = 0.2		β = 0.3		β = 0.4		β = 0.1		β = 0.2		β = 0.3		β = 0.4	
Test Instance	t_{max}	**Best**	**Succ.**	**Cost**	**Succ.**	**Cost**	**Succ.**	**Cost**	**Succ.**	**Cost**	**Succ.**	**Cost**	**Succ.**	**Cost**	**Succ.**	**Cost**	**#Best**	**Cost**
pmed17-p25A	1.3	7317	5	7313.0	8	7315.9	9	7316.2	8	7315.9	6	7312.0	10	7317.0	9	7316.2	8	7315.4
pmed20-p50A	3.8	5872	9	5865.8	10	5872.0	9	5870.7	10	5872.0	8	5869.4	10	5872.0	10	5872.0	10	5872.0
pmed22-p62A	8.3	5995	5	5988.0	4	5985.3	4	5986.0	4	5986.0	4	5984.6	6	5989.4	7	5990.8	5	5988.0
pmed28-p75A	31.6	5681	7	5675.3	8	5677.2	8	5676.8	8	5679.8	5	5672.6	4	5671.9	8	5677.4	8	5677.2
pmed33-p87A	41.1	5793	7	5788.5	5	5785.9	6	5788.1	5	5785.8	9	5790.0	8	5788.5	8	5791.2	8	5790.0
pmed36-p100A	71.0	6461	9	6460.4	8	6459.8	6	6459.0	5	6458.4	10	6461.0	10	6461.0	10	6461.0	10	6461.0
pmed39-p112A	126.0	5935	10	5935.0	10	5935.0	10	5935.0	8	5934.5	10	5935.0	10	5935.0	10	5935.0	10	5935.0
pmed40-p225A	349.6	4571	2	4566.5	2	4563.8	1	4565.6	1	4563.2	3	4566.2	7	4570.5	4	4570.1	3	4570.6
pmed17-p25B	1.1	6905	8	6902.6	10	6905.0	9	6903.8	9	6903.8	7	6901.4	9	6903.8	9	6903.8	9	6903.8
pmed20-p50B	3.6	5665	9	5659.1	10	5665.0	10	5665.0	10	5665.0	9	5641.6	10	5665.0	10	5665.0	10	5665.0
pmed22-p62B	9.1	6259	9	6255.8	9	6255.8	10	6259.0	10	6259.0	10	6259.0	10	6259.0	10	6259.0	10	6259.0
pmed28-p75B	19.2	5642	4	5634.1	2	5632.3	4	5634.3	1	5632.1	8	5640.0	9	5641.0	10	5642.0	8	5640.6
pmed33-p87B	43.3	5840	4	5833.9	3	5832.6	2	5829.6	1	5829.0	5	5833.7	7	5835.2	8	5838.6	10	5840.0
pmed36-p100B	84.3	6219	2	6207.3	2	6210.2	2	6205.4	1	6207.9	1	6199.8	2	6207.3	5	6215.0	5	6214.1
pmed39-p112B	131.0	6198	8	6196.1	7	6197.3	7	6197.3	7	6196.9	7	6196.8	9	6197.8	10	6198.0	10	6198.0
pmed40-p225B	353.0	4528	3	4523.4	1	4520.7	1	4521.5	1	4517.3	1	4521.5	2	4525.8	2	4523.1	3	4525.0
Avg. Values	79.8	5930.1	6.31	5925.3	6.19	5925.9	6.13	5925.8	5.56	5925.4	6.44	5924.0	7.69	5927.5	**8.13**	**5928.6**	7.94	5928.4
Dev. (%)				0.080		0.071		0.071		0.078		0.102		0.043		**0.024**		0.028

The first interesting result of this experiment is that, although LSF produces better results than RLS2 when both of them are executed in isolation, once they are included in a VNS template, the associated performance changes. Specifically, independently of the value of β, VNS variants that consider RLS2 as local search consistently produce better results than those variants that consider LSF. It is worth mentioning that all VNS strategies were executed for the same CPU-time. We can then conclude that the new proposed strategies present a good compromise between quality of solutions and computing time, which makes them as suitable strategies to be integrated into other metaheuristic schemes such as VNS. Among all VNS variants that use RLS2, the one with $\beta = 0.3$ is able to reach the maximum values for Avg. Cost. (5928.6) and Avg. Succ. (8.13/10), together with the minimum Dev. (0.024 %).

We complement this experiment by studying the average convergence of all the VNS variants shown in Table 2 during the first 125 seconds of execution. In order to get useful information, we restrict our attention to the 4 largest test instances, i.e., those with n equal to 112 and 225 (see Table 8). Small ones were not included since the differences among compared methods were imperceptible. As we can observe in Figure 3, all the VNS-RLS2 variants outperform the corresponding VNS-LSF variants. Moreover, in short-term time horizons all VNS-RLS2 variants present similar performance. However, when considering middle or large CPU-times, variants with higher values of β emerge over the lower ones. Specifically, we select the variant with $\beta = 0.3$ for the remaining experiments.

We also tested the effect of the filtering method based on a hash function described in Section 2. This strategy tries to avoid the application of the most computational-demanding procedure (i.e, the local search method) to a solution more than once. To this purpose, we run the best VNS variant (i.e. VNS-RLS2 with $\beta = 0.3$) without this filtering getting an average success rate of 7.88 instead of 8.31 (out of 10) when using this strategy. Hence, we conclude that this filtering method is a good strategy to improve the solution quality avoiding revisiting solutions in a VNS template.

As it was discussed in Section 3, Multi-start VNS may result in an exponential reduction of the error probability with a linear increasing of computational time as the number of independent runs increases. More precisely, we experimentally test if it is better to run a multi-start versions of VNS or a classical implementation of a VNS (starting from a feasible solution and ending when it reaches the termination condition).

In order to test the influence of parameter V over the algorithm performance, different multi-start versions have been executed with values ranging from $V = 1$ to $V = 20$. Since our purpose is to compare the previous best VNS variant with the corresponding multi-start variants (those with $V > 1$), we run them over the test instances for the same t_{max} used in the previous experiment (i.e., the one shown in Table 2). Table 3 shows the results of this experiment by considering $V = \{1, 5, 10, 20\}$. To favor the direct comparison among variants, we replicate again the values of $V = 1$ which correspond to VNS-RLS2 with $\beta = 0.3$ in Table 2. As it can be seen, the success rate increases with V (from 8.13 to 8.81) as the averaged value over all the test instances for $V = 20$. These results show how the convergence behavior and the robustness of the algorithm are improved by using multiple short runs of VNS instead of one long run of a single VNS executed during the same total CPU-time.

Figure 4 shows the average convergence when running all these VNS variants over the largest test instances for different values of V. This figure shows how multi-start VNS methods seems to have slower convergence for higher values of V. Hence, it is necessary to have enough CPU-time to let the algorithm to converge to high quality solutions. In particular, the VNS version with $V = 20$ needs more than 125 seconds to outperform the other ones.

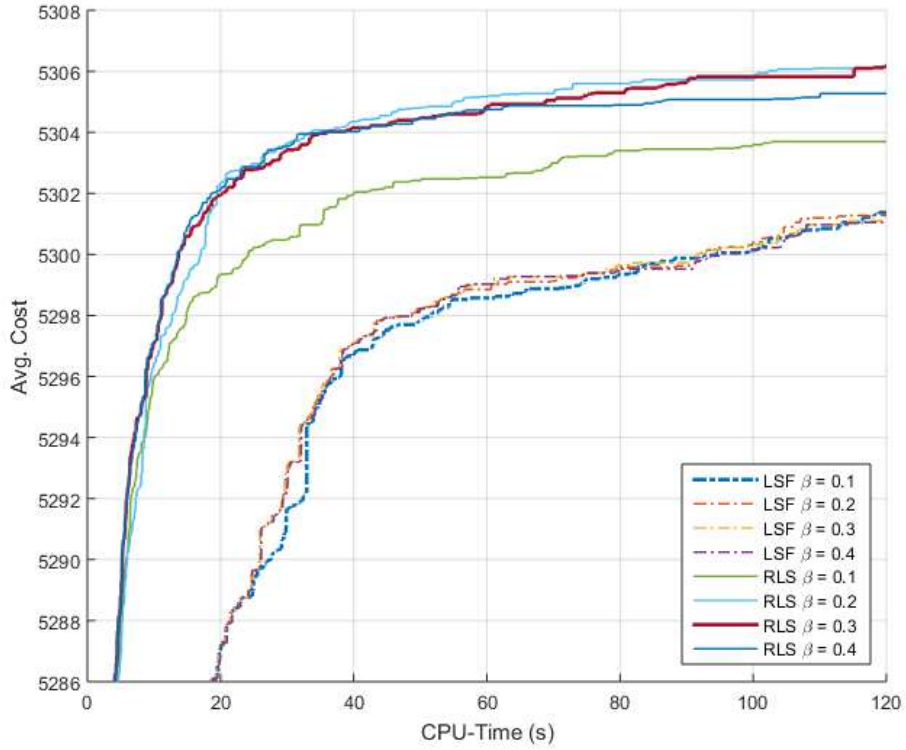

Fig. 3. Average convergence when running the VNS-LSF and VNS-RLS2 algorithms over the larger test instances for different values of β.

Table 3
Performance of Multi-start VNS for several values of V.

		V=1		V=5		V=10		V=20	
Test Instance	**Best**	**Succ.**	**Cost**	**Succ.**	**Cost**	**Succ.**	**Cost**	**Succ.**	**Cost**
pmed17-p25A	7317	9	7316.2	9	7316.7	10	7317.0	10	7317.0
pmed20-p50A	5872	10	5872.0	10	5872.0	10	5872.0	10	5872.0
pmed22-p62A	5995	7	5990.8	10	5995.0	10	5995.0	10	5995.0
pmed28-p75A	5681	8	5677.4	8	5678.2	9	5679.8	10	5681.0
pmed33-p87A	5793	8	5791.2	10	5793.0	10	5793.0	10	5793.0
pmed36-p100A	6461	10	6461.0	10	6461.0	10	6461.0	10	6461.0
pmed39-p112A	5935	10	5935.0	10	5935.0	10	5935.0	10	5935.0
pmed40-p225A	4572	4	4570.1	4	4570.2	3	4570.5	3	4571.1
pmed17-p25B	6905	9	6903.8	10	6905.0	10	6905.0	10	6905.0
pmed20-p50B	5665	10	5665.0	10	5665.0	10	5665.0	10	5665.0
pmed22-p62B	6259	10	6259.0	10	6259.0	10	6259.0	10	6259.0
pmed28-p75B	5642	10	5642.0	10	5642.0	10	5642.0	10	5642.0
pmed33-p87B	5840	8	5838.6	6	5838.7	7	5839.1	10	5840.0
pmed36-p100B	6219	5	6215.0	8	6217.6	7	6217.5	6	6216.8
pmed39-p112B	6198	10	6198.0	10	6198.0	10	6198.0	10	6198.0
pmed40-p225B	4532	2	4523.1	1	4525.9	4	4528.4	2	4527.8
Avg. Values	5930.4	8.13	5928.6	8.50	5929.5	8.75	5929.8	**8.81**	**5929.9**
Dev. (%)			0.029		0.014		0.009		**0.008**

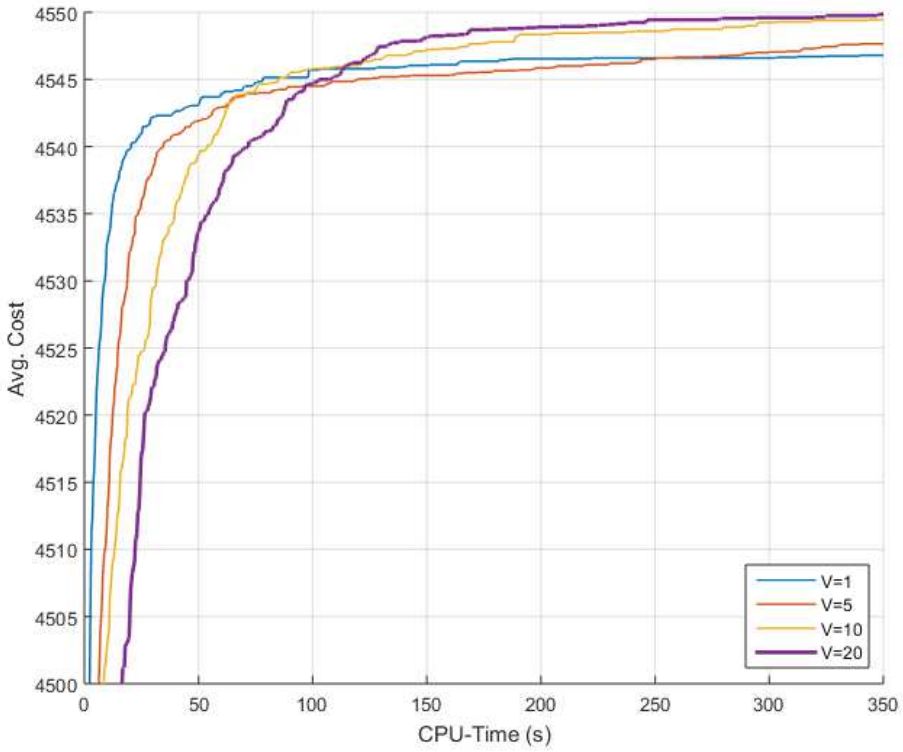

Fig. 4. Average convergence when running the Multi-start VNS algorithm over the larger test instances for different values of V.

In the next experiment, we compare the sequential implementation of multi-start VNS with the parallel versions. Each algorithm was run using the same value of t_{max} as in the previous experiments. We consider an additional method, denoted Multi-start VNS* executed for a computing time proportional to the number of cores used in parallel versions. Specifically, we run it for a computing time 4 (number of cores in our computer) times larger than the one used in parallel versions. Results are shown in Table 4, where we report the same metrics than before (i.e., Avg. Cost, Dev.(%), and Avg. Succ.). We additionally include the average number of iterations (Avg. It.) performed by each algorithm over all the test instances. Note that, for the sake of simplicity, we only report summarized results. In this table, Parallel VNS is the straightforward parallelization of the multi-start versions (without any kind of communication among iterations). Then, as expected, Parallel VNS will perform a larger number of iterations (2926.4 on average) than the sequential version (1473.3), finding better results in terms of average cost (5930.0 vs. 5929.9), average deviation (0.0056 % vs 0.0077 %), and success rare (9.31 vs 8.81). When considering the sequential version executed for longer computing times (Multi-start VNS*), we observe that it presents similar results than Parallel VNS. It is worth mentioning that most of modern processors have several cores that, if algorithms do not use them, they are completely wasted. It is interesting to see

that Multi-start VNS* performs more iterations than Parallel VNS (3236.6 vs. 2926.4). This behavior is probably due to the time overhead needed to manage the threads.

As we described in Section 4, the solution quality of VNS could be further improved by distributing the V solutions into smaller G groups of V/G elements each and allowing the communication between them at every R iterations. To compare the performance of the parallel implementation of the Distributed VNS with the previous sequential multi-start VNS and parallel VNS with $V = 20$, we distribute the 20 solutions into $G = 4$ groups with $V = 5$ solutions each, communicating through FC (DVNS1), RG (DVNS2), and MS (DVNS3) migration topologies every $R = p/5$ iterations. The general conclusion is that all parallel variants improve the results achieved by the sequential VNS, being DVNS3 the best one. It presents the largest average success (9.63/10), averaged cost (5930.2) and lowest average deviation (0.0033%). It is worth mentioning that the migration allows the algorithm to intensify the search around promising regions. Perhaps, for that reason, FC topology used by DVNS1 gets the worst results among them. It may be due to an excess of intensification over the broadcasted solutions to all groups.

Table 4
Performance of Parallel VNS and DVNS against Multi-start VNS.

VNS	G	V	Topology	Avg. It.	Avg. Cost	Dev.(%)	Avg. Succ.
Multi-start VNS	1	20	None	1473.3	5929.9	0.0077	8.81
Multi-start VNS*	1	20	None	3236.6	5930.1	0.0046	9.31
Parallel VNS	4	5	None	2926.4	5930.0	0.0056	9.31
DVNS1	4	5	FC	2773.0	5929.9	0.0073	9.25
DVNS2	4	5	RG	2786.0	5930.1	0.0048	9.50
DVNS3	**4**	**5**	**MS**	**2813.8**	**5930.2**	**0.0032**	**9.63**

5.2. Comparison with state-of-the-art algorithms.

In this section, our best variant, i.e., VNS with RLS2, $\beta = 0.3$ and MS communication topology with $R = p/5$, is compared with previous methods in the state of the art for the OpM in all the 144 available instances. The stopping condition for this experimentation is a time horizon given by a maximum number of iterations. After some testing, it has been set to $20{\cdot}p$ for each instance. We have observed that this value is sufficient to get high quality results. We also tested $100{\cdot}p$ and a larger parallel version with $V = 25$ instead of $V = 4$, but these variants produce marginal improvements with a considerably larger computing time.

The best identified procedures in the related literature include two variants of Branch & Cut, one standard (BC) and another one coupled with the Tabu Search method (XTS), both presented in (Belotti et al., 2007); and a recent Greedy Randomized Adaptive Search Procedure (GRASP) presented in (Colmenar et al., 2016). The BC is an exact procedure that, executed for unlimited time, guarantees the optimality of the solution found. However, considering that we are proposing heuristic procedures, we set the maximum computing time of this exact method to 3600 seconds. If after that time the BC has not found the optimal solution, we interrupt its execution, returning the best solution found. The XTS is configured with the best parameters described in Belotti et al. (2007). We also include in the comparison the best GRASP variant reported in Colmenar et al. (2016), i.e., constructive $C2$ with $\alpha = 0.75$, local search LSF, incremental computation of the cost, and filter strategy with $q = 2$, executed for a time horizon of 1000 iterations.

Table 5 reports the results showing the merit of the VNS approach over the previous state-of-the-art procedures. Again, we report similar metrics than the ones used in previous experiments. Results of BC method have been also included as a baseline for the comparison (i.e., not to compare the BC itself with other heuristics). VNS clearly outperforms previous algorithms in all the considered metrics. It obtains the best results in 137 instances (out of 144), while the competitors obtain 23 (XTS) and 92 (GRASP). In addition, VNS obtains these results by employing only 10% the total CPU-time consumed by the previous heuristics. In order to facilitate the direct comparison among methods, results are divided according to the number of facilities of the instances used in the experimentation, p. Specifically, we identify the sets *Small* ($p = n/16$), *Medium* ($p = n/8$), and *Large* ($p = n/4$), containing 48 instances each one. We can see that in the *Small*, the best performing methods (i.e., GRASP and VNS) achieve the same results in terms of quality. Therefore, in our opinion this set should not be considered for future comparisons since the only perceptible challenge seems to be the reduction of the computing time.

In *Medium* and *Large* sets we can observe relevant differences among the compared methods. VNS emerges as the best method since it does not match the best solution in only 5 and 2 instances, respectively. In addition, it employs short computing times to obtain those results. It is worth mentioning that OpM instances used in this experimentation have large values of the objective function (see the column Avg. Cost). Therefore, improving the value of the cost in hundreds barely affects to the value of the average deviation. It is also important to remark that the OpM problem consists in maximizing a minimum value. Then, these problems become in an actual challenge for heuristics methods. Therefore, the reduction from 1.84% (XTS) and 0.05% (GRASP) to 0.001 % (VNS) can be considered as a success.

Table 5
Comparison of VNS against the previous state of the art for OpM.

Size	Algorithm	Avg. Cost	Dev. (%)	Avg. Time	#Best
Small	BC	7452.0	1.7195	1800.0	9
	XTS	7463.0	1.5747	272.4	16
	GRASP	7582.4	0.0000	247.7	48
	VNS	**7582.4**	**0.0000**	**8.4**	**48**
Medium	BC	5784.8	1.2296	6915.7	6
	XTS	5814.5	0.7220	397.2	6
	GRASP	5855.5	0.0208	406.0	35
	VNS	**5856.6**	**0.0025**	**28.6**	**43**
Large	BC	4090.3	2.9134	7301.1	1
	XTS	4169.8	1.0265	331.7	1
	GRASP	4205.0	0.1913	642.4	9
	VNS	**4213.1**	**0.0002**	**108.9**	**46**
Full set	BC	5775.7	1.8425	5338.9	16
	XTS	5815.8	1.1615	333.8	23
	GRASP	5881.0	0.0532	432.0	92
	VNS	**5884.0**	**0.0013**	**48.7**	**137**

In order to support the previous assessments, we applied a Friedman test to the results obtained by all the heuristics analyzed in this paper (XTS, GRASP and VNS) over the full set of instances. This non-parametric statistical test is similar to the repeated measures of ANOVA test, and it is usually used to detect differences in algorithm performance across the same set of instances. The test ranks each method on each instance, and finally considers the values of the ranks obtained by each algorithm. These results were: 2.56 (XTS), 2.25 (GRASP), and 1.19 (VNS). The average ranking for our VNS method (close to 1) reflects that it generally obtains higher-quality solutions. Finally, the obtained p-value (< 0.0001)

clearly indicates that there is enough statistical evidence to confirm that there are differences among the three algorithms.

Having confirmed the existence of differences between the methods, we conducted two well-known non-parametric tests for pairwise comparisons, the Wilcoxon test and the Sign test, in order to detect consistent differences between VNS and the two previous methods. The Wilcoxon test answers the question: Do the two samples (solutions obtained with both methods in our case) represent two different populations? The resulting p-value < 0.0001 clearly indicates that the values compared come from different methods (using a typical significance level $\alpha = 0.05$ as the threshold for rejecting the null hypothesis). On the other hand, the Sign test computes the number of instances in which an algorithm outperforms the competitor (*sign*). Table 6 summarize the results of the tests. Given the low p-values, the statistical tests clearly support the superiority of VNS over the XTP algorithm described in Belotti et al. (2007), and the GRASP procedure proposed in Colmenar et al. (2016).

Table 6
Results of the non-parametric Wilcoxon and Sign test applied to results given by XTS, GRASP and VNS heuristics.

	Wilcoxon	**Sign test**	
Compared algorithms	*p-value*	*sign*	*p-value*
VNS vs XTS	< 0.0001	120	< 0.0001
VNS vs GRASP	< 0.0001	52	< 0.0001

Finally, in order to contribute to a further research in this problem, we detail in Table 7 the results of our experiments breaking down them into the instance level. Table 7 shows the best cost found for each one of the 144 instances we have dealt with. In addition, it is also indicated the algorithm/s that obtained the best solution, as well as the time spent to reach it. In the cases where two or more algorithms found the same best solution, the reported time corresponds to the faster one, which is shown in bold font.

6. Conclusions

In this work we have studied the application of a VNS approach to solve the OpM problem. In particular, we have developed two local search algorithms based on the exploration of a reduced neighborhood which gives a good compromise between the quality of solutions and the computation time, making these strategies suitable to be integrated into a higher level metaheuristic such as VNS. The performance of the local search stage in the VNS algorithm has been compared with previous approaches, showing an increase in the quality of solutions of 50% and 25% in short and large computing times. We have developed a parallel version of the VNS algorithm that executes multiple single VNS algorithms distributed among several processors, communicating to each other at certain iterations according to three different exchanging topologies, which we have been also compared among them.

Before engaging into competitive testing, we performed a series of preliminary tests to determine the contribution of the various elements that we have designed, as the perturbation depth of the shaking mechanism or the best interaction schema in the parallel version. Therefore, a final configuration for the VNS scheme has been defined according to the results of these experiments.

The extensive final comparison between the proposed parallel VNS algorithm and the two best methods identified in the literature, XTS and GRASP, reveals that our algorithm is able to outperform the current state-of-the-art methods in both short and long time horizons. In particular, our approach finds

Table 7
Best cost obtained by each one of the instances, algorithms that obtained the value, and time (in seconds) spent by the faster algorithm (in bold).

Instance	Best	Method	Time	Instance	Best	Method	Time
pmed17-p25.A.txt	7317	GRASP; **VNS**	1.9	pmed17-p25.B.txt	6905	GRASP; **VNS**	1.7
pmed17-p50.A.txt	5411	VNS	5.8	pmed17-p50.B.txt	5563	BC; XTS; GRASP; **VNS**	5.0
pmed17-p100.A.txt	4054	GRASP; **VNS**	18.6	pmed17-p100.B.txt	3992	GRASP; **VNS**	18.0
pmed18-p25.A.txt	7432	GRASP; **VNS**	1.7	pmed18-p25.B.txt	7662	BC; XTS; GRASP; **VNS**	1.7
pmed18-p50.A.txt	5746	BC; XTS; GRASP; **VNS**	5.0	pmed18-p50.B.txt	5852	BC; XTS; GRASP; **VNS**	4.8
pmed18-p100.A.txt	4220	BC; XTS; **VNS**	17.3	pmed18-p100.B.txt	4122	VNS	16.8
pmed19-p25.A.txt	7020	GRASP; **VNS**	1.7	pmed19-p25.B.txt	6816	BC; XTS; GRASP; **VNS**	1.7
pmed19-p50.A.txt	5387	GRASP; **VNS**	4.8	pmed19-p50.B.txt	5423	GRASP; **VNS**	4.9
pmed19-p100.A.txt	4033	GRASP; **VNS**	17.6	pmed19-p100.B.txt	4016	GRASP; **VNS**	16.7
pmed20-p25.A.txt	7648	GRASP; **VNS**	1.6	pmed20-p25.B.txt	7349	XTS; GRASP; **VNS**	1.6
pmed20-p50.A.txt	5872	GRASP; **VNS**	4.8	pmed20-p50.B.txt	5665	BC; XTS; GRASP; **VNS**	4.8
pmed20-p100.A.txt	4063	VNS	18.0	pmed20-p100.B.txt	4067	VNS	16.4
pmed21-p31.A.txt	7304	GRASP; **VNS**	3.1	pmed21-p31.B.txt	7331	BC; XTS; GRASP; **VNS**	3.2
pmed21-p62.A.txt	5784	GRASP; **VNS**	9.9	pmed21-p62.B.txt	5870	GRASP; **VNS**	9.7
pmed21-p125.A.txt	4155	VNS	38.4	pmed21-p125.B.txt	4033	GRASP; **VNS**	35.8
pmed22-p31.A.txt	7900	GRASP; **VNS**	3.2	pmed22-p31.B.txt	7695	BC; XTS; GRASP; **VNS**	3.2
pmed22-p62.A.txt	5995	GRASP; **VNS**	9.9	pmed22-p62.B.txt	6259	GRASP; **VNS**	10.1
pmed22-p125.A.txt	4358	VNS	35.6	pmed22-p125.B.txt	4338	VNS	39.2
pmed23-p31.A.txt	7841	GRASP; **VNS**	3.0	pmed23-p31.B.txt	7137	BC; XTS; GRASP; **VNS**	3.2
pmed23-p62.A.txt	5785	GRASP; **VNS**	10.2	pmed23-p62.B.txt	5724	GRASP; **VNS**	9.7
pmed23-p125.A.txt	4114	VNS	36.7	pmed23-p125.B.txt	4095	VNS	37.4
pmed24-p31.A.txt	7425	GRASP; **VNS**	3.1	pmed24-p31.B.txt	7190	GRASP; **VNS**	3.1
pmed24-p62.A.txt	5528	GRASP; **VNS**	10.3	pmed24-p62.B.txt	5752	GRASP; **VNS**	10.1
pmed24-p125.A.txt	4091	VNS	37.0	pmed24-p125.B.txt	4072	VNS	36.4
pmed25-p31.A.txt	7552	GRASP; **VNS**	3.2	pmed25-p31.B.txt	7552	GRASP; **VNS**	3.2
pmed25-p62.A.txt	5767	GRASP; **VNS**	9.7	pmed25-p62.B.txt	5692	GRASP; **VNS**	10.0
pmed25-p125.A.txt	4155	VNS	35.5	pmed25-p125.B.txt	4233	VNS	40.4
pmed26-p37.A.txt	8112	GRASP; **VNS**	5.5	pmed26-p37.B.txt	7643	XTS; GRASP	5.5
pmed26-p75.A.txt	5789	GRASP; **VNS**	17.9	pmed26-p75.B.txt	5923	GRASP; **VNS**	18.3
pmed26-p150.A.txt	4341	VNS	76.7	pmed26-p150.B.txt	4173	VNS	70.3
pmed27-p37.A.txt	7556	GRASP; **VNS**	5.8	pmed27-p37.B.txt	7448	BC; XTS; GRASP; **VNS**	5.4
pmed27-p75.A.txt	5668	GRASP; **VNS**	18.4	pmed27-p75.B.txt	5844	VNS	19.6
pmed27-p150.A.txt	4062	VNS	68.7	pmed27-p150.B.txt	4144	VNS	67.1
pmed28-p37.A.txt	7366	GRASP; **VNS**	5.8	pmed28-p37.B.txt	7388	XTS; GRASP; **VNS**	5.5
pmed28-p75.A.txt	5681	VNS	18.7	pmed28-p75.B.txt	5642	GRASP; **VNS**	19.2
pmed28-p150.A.txt	4099	VNS	69.7	pmed28-p150.B.txt	4069	GRASP; **VNS**	82.1
pmed29-p37.A.txt	7404	GRASP; **VNS**	5.5	pmed29-p37.B.txt	7529	GRASP; **VNS**	5.6
pmed29-p75.A.txt	5880	BC; XTS; GRASP; **VNS**	18.7	pmed29-p75.B.txt	5709	GRASP; **VNS**	19.2
pmed29-p150.A.txt	4141	VNS	62.5	pmed29-p150.B.txt	4157	GRASP; **VNS**	65.5
pmed30-p37.A.txt	7704	GRASP; **VNS**	5.7	pmed30-p37.B.txt	8048	BC; XTS; GRASP; **VNS**	5.4
pmed30-p75.A.txt	6189	GRASP; **VNS**	19.3	pmed30-p75.B.txt	6041	GRASP; **VNS**	18.5
pmed30-p150.A.txt	4385	VNS	69.6	pmed30-p150.B.txt	4313	VNS	75.3
pmed31-p43.A.txt	7424	GRASP; **VNS**	9.0	pmed31-p43.B.txt	7320	XTS; GRASP; **VNS**	9.1
pmed31-p87.A.txt	5905	BC; XTS; GRASP; **VNS**	32.1	pmed31-p87.B.txt	5618	VNS	31.3
pmed31-p175.A.txt	4135	VNS	124.7	pmed31-p175.B.txt	4138	VNS	124.4
pmed32-p43.A.txt	7794	GRASP; **VNS**	9.4	pmed32-p43.B.txt	7899	BC; XTS; GRASP; **VNS**	8.8
pmed32-p87.A.txt	5925	VNS	32.3	pmed32-p87.B.txt	5852	GRASP; **VNS**	31.7
pmed32-p175.A.txt	4242	VNS	111.6	pmed32-p175.B.txt	4244	VNS	126.9
pmed33-p43.A.txt	7598	GRASP; **VNS**	9.1	pmed33-p43.B.txt	7611	GRASP; **VNS**	8.7
pmed33-p87.A.txt	5793	VNS	29.6	pmed33-p87.B.txt	5840	GRASP; **VNS**	32.1
pmed33-p175.A.txt	4105	VNS	129.8	pmed33-p175.B.txt	4156	VNS	112.5
pmed34-p43.A.txt	7725	GRASP; **VNS**	8.9	pmed34-p43.B.txt	7514	BC; XTS; GRASP; **VNS**	9.2
pmed34-p87.A.txt	5849	GRASP; **VNS**	31.2	pmed34-p87.B.txt	5857	GRASP; **VNS**	30.9
pmed34-p175.A.txt	4287	VNS	69.5	pmed34-p175.B.txt	4270	GRASP; **VNS**	136.5
pmed35-p50.A.txt	7155	GRASP; **VNS**	15.8	pmed35-p50.B.txt	7570	XTS; GRASP; **VNS**	16.0
pmed35-p100.A.txt	5845	GRASP; **VNS**	54.1	pmed35-p100.B.txt	5639	VNS	52.8
pmed35-p200.A.txt	4007	GRASP; **VNS**	167.0	pmed35-p200.B.txt	4109	VNS	206.6
pmed36-p50.A.txt	8179	GRASP; **VNS**	15.2	pmed36-p50.B.txt	8144	GRASP; **VNS**	14.9
pmed36-p100.A.txt	6461	GRASP; **VNS**	52.9	pmed36-p100.B.txt	6219	VNS	54.7
pmed36-p200.A.txt	4319	VNS	202.1	pmed36-p200.B.txt	4319	VNS	199.0
pmed37-p50.A.txt	7830	GRASP; **VNS**	15.5	pmed37-p50.B.txt	8379	GRASP; **VNS**	14.8
pmed37-p100.A.txt	6203	GRASP; **VNS**	50.8	pmed37-p100.B.txt	6209	VNS	49.9
pmed37-p200.A.txt	4593	VNS	184.1	pmed37-p200.B.txt	4609	VNS	201.8
pmed38-p56.A.txt	7432	GRASP; **VNS**	23.4	pmed38-p56.B.txt	7535	GRASP; **VNS**	24.5
pmed38-p112.A.txt	5913	VNS	86.1	pmed38-p112.B.txt	5949	GRASP; **VNS**	84.5
pmed38-p225.A.txt	4428	VNS	291.1	pmed38-p225.B.txt	4446	VNS	319.1
pmed39-p56.A.txt	7712	GRASP; **VNS**	22.7	pmed39-p56.B.txt	7625	XTS; GRASP; **VNS**	23.6
pmed39-p112.A.txt	5935	GRASP; **VNS**	78.0	pmed39-p112.B.txt	6198	GRASP; **VNS**	78.7
pmed39-p225.A.txt	4369	VNS	322.2	pmed39-p225.B.txt	4266	VNS	328.5
pmed40-p56.A.txt	8211	GRASP; **VNS**	22.8	pmed40-p56.B.txt	8022	XTS; GRASP; **VNS**	24.4
pmed40-p112.A.txt	6272	GRASP; **VNS**	79.0	pmed40-p112.B.txt	6200	VNS	72.5
pmed40-p225.A.txt	4571	VNS	285.0	pmed40-p225.B.txt	4524	VNS	271.6

the best known results in previously used instances in considerably shorter computing times. The superiority of our method is further supported by the low p-values associated with non-parametric tests for detecting statistical significant differences between the algorithms.

Future research on the problem will examine other complex metaheuristics such as Simulated Annealing or any other hybrid approaches as variants of the VNS proposed here. Besides, the VNS approach could be applied to a multi-objective optimization scenario in the OpM problem.

7. Appendix

We have used a set of instances previously described in (Colmenar et al., 2016). In particular, they were generated by considering 24 instances (from `pmed17` to `pmed40`) of the well know p-median problem[1], where the number of nodes ranges from 400 to 900. In order to transform a p-median instance into an obnoxious p-median one, (Belotti et al., 2007) described the following procedure. Given the original instance with n nodes, this method selects $n/2$ nodes at random to be the set of clients. The remaining $n/2$ become the set of facilities. Additionally, for each original instance, (Belotti et al., 2007) derived three new instances for the OpM by considering three values of p: $\lfloor n/4 \rfloor$, $\lfloor n/8 \rfloor$ and $\lfloor n/16 \rfloor$. Table 8 reports the main characteristics of the new set of of 144 instances, where $|I|/|J|$ represents the number of clients/facilities, and p the number of required facilities.

Acknowledgements

This work has been partially supported by the Spanish Government Minister of Science and Innovation under grants MINECO/FEDER TIN2015-69542-C2-1-R, TIN2014-54806-R and TIN2015-65460-C2.

References

E. Alba and J.M. Troya. Influence of the migration policy in parallel distributed gas with structured and panmictic populations. *Applied Intelligence*, 12(3):163–181, 2000. ISSN 1573-7497. .

E. Alba and M. Troya. A survey of parallel distributed genetic algorithms. *Complexity*, 4(4):31–52, 1999.

Z. Amirgaliyeva, N. Mladenovic, R. Todosijevic, and D. Urosevic. Solving the maximum min-sum dispersion by alternating formulations of two different problems. *European Journal of Operational Research*, pages –, 2016. ISSN 0377-2217.

R. Azencott. *Sequential simulated annealing: speed of convergence and acceleration techniques*. Wiley-Interscience series in discrete mathematics. New York: John Wiley and Sons, 1992. ISBN 9780471532316.

J. E. Beasley. OR-Library: distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41(11): 1069–1072, 1990.

P. Belotti, M Labbé, F Maffioli, and M. Ndiaye. A branch-and-cut method for the obnoxious p-median problem. *4OR*, 5(4): 299–314, 2007. ISSN 1619-4500.

D.R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley professional computing series. Addison-Wesley, 1997.

R.L. Church and R.S. Garfinkel. Locating an obnoxious facility on a network. *Trans. Sci.*, 12(2):107–118, 1978.

J.M. Colmenar, P. Greistorfer, R. Martí, and A. Duarte. Advanced greedy randomized adaptive search procedure for the obnoxious p-median problem. *European Journal of Operational Research*, 252:432–442, 2016. ISSN 0377-2217. .

[1] http://people.brunel.ac.uk/~mastjjb/jeb/orlib/pmedinfo.html

Table 8
Instances generated from the OR-Library (Beasley, 1990).

N°	Instance	n	$\|I\|/\|J\|$	p	N°	Instance	n	$\|I\|/\|J\|$	p
1	**pmed17-p25.A.txt**	**400**	**200**	**25**	**73**	**pmed17-p25.B.txt**	**400**	**200**	**25**
2	pmed17-p50.A.txt	400	200	50	74	pmed17-p50.B.txt	400	200	50
3	pmed17-p100.A.txt	400	200	100	75	pmed17-p100.B.txt	400	200	100
4	pmed18-p25.A.txt	400	200	25	76	pmed18-p25.B.txt	400	200	25
5	pmed18-p50.A.txt	400	200	50	77	pmed18-p50.B.txt	400	200	50
6	pmed18-p100.A.txt	400	200	100	78	pmed18-p100.B.txt	400	200	100
7	pmed19-p25.A.txt	400	200	25	79	pmed19-p25.B.txt	400	200	25
8	pmed19-p50.A.txt	400	200	50	80	pmed19-p50.B.txt	400	200	50
9	pmed19-p100.A.txt	400	200	100	81	pmed19-p100.B.txt	400	200	100
10	pmed20-p25.A.txt	500	250	25	82	pmed20-p25.B.txt	500	250	25
11	**pmed20-p50.A.txt**	**500**	**250**	**50**	**83**	**pmed20-p50.B.txt**	**500**	**250**	**50**
12	pmed20-p100.A.txt	500	250	100	84	pmed20-p100.B.txt	500	250	100
13	pmed21-p31.A.txt	500	250	31	85	pmed21-p31.B.txt	500	250	31
14	pmed21-p62.A.txt	500	250	62	86	pmed21-p62.B.txt	500	250	62
15	pmed21-p125.A.txt	500	250	125	87	pmed21-p125.B.txt	500	250	125
16	pmed22-p31.A.txt	500	250	31	88	pmed22-p31.B.txt	500	250	31
17	**pmed22-p62.A.txt**	**500**	**250**	**62**	**89**	**pmed22-p62.B.txt**	**500**	**250**	**62**
18	pmed22-p125.A.txt	500	250	125	90	pmed22-p125.B.txt	500	250	125
19	pmed23-p31.A.txt	500	250	31	91	pmed23-p31.B.txt	500	250	31
20	pmed23-p62.A.txt	500	250	62	92	pmed23-p62.B.txt	500	250	62
21	pmed23-p125.A.txt	500	250	125	93	pmed23-p125.B.txt	500	250	125
22	pmed24-p31.A.txt	500	250	31	94	pmed24-p31.B.txt	500	250	31
23	pmed24-p62.A.txt	500	250	62	95	pmed24-p62.B.txt	500	250	62
24	pmed24-p125.A.txt	500	250	125	96	pmed24-p125.B.txt	500	250	125
25	pmed25-p31.A.txt	500	250	31	97	pmed25-p31.B.txt	500	250	31
26	pmed25-p62.A.txt	500	250	62	98	pmed25-p62.B.txt	500	250	62
27	pmed25-p125.A.txt	500	250	125	99	pmed25-p125.B.txt	500	250	125
28	pmed26-p37.A.txt	600	300	37	100	pmed26-p37.B.txt	600	300	37
29	pmed26-p75.A.txt	600	300	75	101	pmed26-p75.B.txt	600	300	75
30	pmed26-p150.A.txt	600	300	150	102	pmed26-p150.B.txt	600	300	150
31	pmed27-p37.A.txt	600	300	37	103	pmed27-p37.B.txt	600	300	37
32	pmed27-p75.A.txt	600	300	75	104	pmed27-p75.B.txt	600	300	75
33	pmed27-p150.A.txt	600	300	150	105	pmed27-p150.B.txt	600	300	150
34	pmed28-p37.A.txt	600	300	37	106	pmed28-p37.B.txt	600	300	37
35	**pmed28-p75.A.txt**	**600**	**300**	**75**	**107**	**pmed28-p75.B.txt**	**600**	**300**	**75**
36	pmed28-p150.A.txt	600	300	150	108	pmed28-p150.B.txt	600	300	150
37	pmed29-p37.A.txt	600	300	37	109	pmed29-p37.B.txt	600	300	37
38	pmed29-p75.A.txt	600	300	75	110	pmed29-p75.B.txt	600	300	75
39	pmed29-p150.A.txt	600	300	150	111	pmed29-p150.B.txt	600	300	150
40	pmed30-p37.A.txt	600	300	37	112	pmed30-p37.B.txt	600	300	37
41	pmed30-p75.A.txt	600	300	75	113	pmed30-p75.B.txt	600	300	75
42	pmed30-p150.A.txt	600	300	150	114	pmed30-p150.B.txt	600	300	150
43	pmed31-p43.A.txt	700	350	43	115	pmed31-p43.B.txt	700	350	43
44	pmed31-p87.A.txt	700	350	87	116	pmed31-p87.B.txt	700	350	87
45	pmed31-p175.A.txt	700	350	175	117	pmed31-p175.B.txt	700	350	175
46	pmed32-p43.A.txt	700	350	43	118	pmed32-p43.B.txt	700	350	43
47	pmed32-p87.A.txt	700	350	87	119	pmed32-p87.B.txt	700	350	87
48	pmed32-p175.A.txt	700	350	175	120	pmed32-p175.B.txt	700	350	175
49	pmed33-p43.A.txt	700	350	43	121	pmed33-p43.B.txt	700	350	43
50	**pmed33-p87.A.txt**	**700**	**350**	**87**	**122**	**pmed33-p87.B.txt**	**700**	**350**	**87**
51	pmed33-p175.A.txt	700	350	175	123	pmed33-p175.B.txt	700	350	175
52	pmed34-p43.A.txt	700	350	43	124	pmed34-p43.B.txt	700	350	43
53	pmed34-p87.A.txt	700	350	87	125	pmed34-p87.B.txt	700	350	87
54	pmed34-p175.A.txt	700	350	175	126	pmed34-p175.B.txt	700	350	175
55	pmed35-p50.A.txt	800	400	50	127	pmed35-p50.B.txt	800	400	50
56	pmed35-p100.A.txt	800	400	100	128	pmed35-p100.B.txt	800	400	100
57	pmed35-p200.A.txt	800	400	200	129	pmed35-p200.B.txt	800	400	200
58	pmed36-p50.A.txt	800	400	50	130	pmed36-p50.B.txt	800	400	50
59	**pmed36-p100.A.txt**	**800**	**400**	**100**	**131**	**pmed36-p100.B.txt**	**800**	**400**	**100**
60	pmed36-p200.A.txt	800	400	200	132	pmed36-p200.B.txt	800	400	200
61	pmed37-p50.A.txt	800	400	50	133	pmed37-p50.B.txt	800	400	50
62	pmed37-p100.A.txt	800	400	100	134	pmed37-p100.B.txt	800	400	100
63	pmed37-p200.A.txt	800	400	200	135	pmed37-p200.B.txt	800	400	200
64	pmed38-p56.A.txt	900	450	56	136	pmed38-p56.B.txt	900	450	56
65	pmed38-p112.A.txt	900	450	112	137	pmed38-p112.B.txt	900	450	112
66	pmed38-p225.A.txt	900	450	225	138	pmed38-p225.B.txt	900	450	225
67	pmed39-p56.A.txt	900	450	56	139	pmed39-p56.B.txt	900	450	56
68	**pmed39-p112.A.txt**	**900**	**450**	**112**	**140**	**pmed39-p112.B.txt**	**900**	**450**	**112**
69	pmed39-p225.A.txt	900	450	225	141	pmed39-p225.B.txt	900	450	225
70	pmed40-p56.A.txt	900	450	56	142	pmed40-p56.B.txt	900	450	56
71	pmed40-p112.A.txt	900	450	112	143	pmed40-p112.B.txt	900	450	112
72	**pmed40-p225.A.txt**	**900**	**450**	**225**	**144**	**pmed40-p225.B.txt**	**900**	**450**	**225**

T.G. Crainic, M. Gendreau, P. Hansen, and N. Mladenovic. Cooperative parallel variable neighborhood search for the p-median. *Journal of Heuristics*, 10(3):293–314, 2004.

J. Current, M. Daskin, and D. Shilling. Discrete network location models. In Zvi Drezner and Horst W. Hamacher, editors, *Facility Location: Applications and Theory*, Springer series in operations research, chapter 3, pages 83–120. Springer Science & Business Media, 2004.

L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

M. Defersha, F.M.and Chen. A parallel multiple markov chain simulated annealing for multi-period manufacturing cell formation problems. *The International Journal of Advanced Manufacturing Technology*, 37(1):140–156, 2008.

C. Defryn and K. Sorensen. A fast two-level variable neighborhood search for the clustered vehicle routing problem. *Computers & Operations Research*, 83:78 – 94, 2017. ISSN 0305-0548.

A. Duarte, J. J. Pantrigo, E.G. Pardo, and J. Snchez-Oro. Parallel variable neighbourhood search strategies for the cutwidth minimization problem. *IMA Journal of Management Mathematics*, 27(1):55, 2016.

E. Erkut and S. Neuman. Analytical models for locating undesirable facilities. *EJOR*, 40(3):275–291, 1989.

R.L. Francis and J.A. White. *Facility layout and location: an analytical approach*. International Industrial and Systems Engineering Series. Prentice-Hall, 1974. ISBN 9780132991490.

F. García-López, B. Melián-Batista, J.A. Moreno-Pérez, and J.M. Moreno-Vega. The parallel variable neighborhood search for the p-median problem. *Journal of Heuristics*, 8(3):375–388, 2002. ISSN 1572-9397. .

S.L. Hakimi. Optimum locations of switching centers and the absolute centers and medians of a graph. *Ops Res.*, 12(3): 450–459, 1964.

P. Hansen and N. Mladenovic. Variable neighborhood search: Principles and applications. *European Journal of Operational Research*, 130(3):449 – 467, 2001. ISSN 0377-2217. .

P. Hansen, N. Mladenovic, and J.A. Moreno Pérez. Variable neighbourhood search: methods and applications. *4OR*, 6(4): 319–360, 2008. ISSN 1614-2411. .

P. Hansen, N. Mladenovic, and J.A. Moreno Pérez. Variable neighbourhood search: methods and applications. *Annals of Operations Research*, 175(1):367–407, 2010. ISSN 1572-9338. .

M.J. Kuby. Programming models for facility dispersion: the p-dispersion and maxisum dispersion problems. *Mathematical and Computer Modelling*, 10(10):792 –, 1988. ISSN 0895-7177. .

R. Martí, R. Aceves, M.T. León, M. Moreno-Vega, and A. Duarte. *Intelligent multi-start methods, Handbook of Metaheuristics 3rd edition (Forthcoming)*. Springer, 2017.

M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

B. Menéndez, J. Sánchez-Oro, E.G. Pardo, and A. Duarte. Parallel variable neighborhood search for the min-max order batching problem. *International Transaction of Operational Research*, 24, 2017. .

J.R. Norris. *Markov Chains*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 1998. ISBN 9781107393479.

M. Nowostawski and R. Poli. Parallel genetic algorithm taxonomy. In *Knowledge-Based Intelligent Information Engineering Systems, 1999. Third International Conference*, pages 88–92. IEEE, 1999.

NVIDIA Corporation. CUDA Zone website. https://developer.nvidia.com/cuda-zone, 2017. Online; accessed February 2017.

E.G. Pardo, N. Mladenovic, J.J. Pantrigo, and A. Duarte. Variable formulation search for the cutwidth minimization problem. *Applied Soft Computing*, 13(5):2242 – 2252, 2013.

B. Rosa, M. Souza, R. de Souza, M. de Frana Filho, Z. Ales, and P. Michelon. Algorithms for job scheduling problems with distinct time windows and general earliness/tardiness penalties. *Computers & Operations Research*, 81:203 – 215, 2017.

J. Sánchez-Oro, M. Sevaux, A. Rossi, R. Martí, and A. Duarte. Solving dynamic memory allocation problems in embedded systems with parallel variable neighborhood search strategies. *Electronic Notes in Discrete Mathematics*, 47:85–92, 2015.

J. Sánchez-Oro, M. Sevaux, A. Rossi, R. Martí, and A. Duarte. Improving performance of embedded systems with variable neighborhood search. *Applied Soft Computing*, 53:217–226, 2017.

D. Shier. A min-max theorem for p-center problems on a tree. *Transportation Science*, 11(3):243–252, 1977. ISSN 1526-5447.

Arie Tamir. Obnoxious facility location on graphs. *SIAM J. Discret. Math.*, 4(4):550–567, September 1991. ISSN 0895-4801. . URL http://dx.doi.org/10.1137/0404048.

S. S. Ting. *Obnoxious facility location problems on networks*. PhD thesis, The Johns Hopkins University, 1988.