IMA Journal of Management Mathematics (2013) Page 1 of 19
doi:10.1093/imaman/dpt026

Parallel variable neighbourhood search strategies for the cutwidth minimization problem

Abraham Duarte*, Juan J. Pantrigo, Eduardo G. Pardo and Jesús Sánchez-Oro
Dept. Ciencias de la Computación, Universidad Rey Juan Carlos, Móstoles, Spain
*Corresponding author: abraham.duarte@urjc.es juanjose.pantrigo@urjc.es
eduardo.pardo@urjc.es jesus.sanchezoro@urjc.es

[Received on 31 January 2013; accepted on 17 November 2013]

Variable neighbourhood search (VNS) and all its variants have been successfully proved in hard combinatorial optimization problems. However, there are only few works concerning parallel VNS algorithms, compared with the amount of works devoted to sequential VNS design. In this paper, we propose different parallel designs for the VNS schema. We illustrate the performance of these general strategies by parallelizing a new VNS variant called variable formulation search (VFS). Specifically, we propose six different variants which differ in the VNS stages to be parallelized as well as in the communication mechanisms among processes. We group these variants into three different templates. 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 neighbourhoods. We test the resulting designs on the cutwidth minimization problem (CMP). Experimental results show that the parallel implementation of the VFS outperforms previous methods in the state of the art for the CMP. This fact is also confirmed by non-parametric statistical tests.

Keywords: variable neighbourhood search; variable formulation search; parallel designs; cutwidth minimization problem.

1. Introduction

Metaheuristics (MHs) are among the most prominent and successful techniques to solve a large amount of complex and computationally hard combinatorial and numerical optimization problems arising in human activities, such as economics (e.g. portfolio selection), industry (e.g. scheduling or logistics) or engineering (e.g. routing), among many others (Gendreau & Potvin, 2010). MHs can be seen as general algorithmic frameworks that require relatively few modifications to be adapted to tackle a specific problem. They constitute a very diverse family of optimization algorithms including methods such as simulated annealing, tabu search, multi-start methods, iterated local search, variable neighbourhood search (VNS), greedy randomized adaptive search procedure (GRASP), memetic algorithms, scatter search, evolutionary algorithms or ant colony optimization.

The VNS and all its variants have proved to be very successful in hard optimization problems. This MH was originally proposed by Mladenović & Hansen (1997). It is based on the exploration of a dynamic neighbourhood model. Contrary to other trajectory-based MHs, VNS allows changes of the neighbourhood structure along the search. In particular, VNS explores increasingly neighbourhoods of the current best found solution. The basic idea of the VNS is to change the neighbourhood structure when the local search is trapped on a local optimum.

Considering that VNS is a relatively new MH, it has not yet been investigated much from a parallelization point of view. As per the authors' knowledge, there are only two relevant papers reported in

the literature on the parallelization of VNS. Specifically, García-López *et al.* (2002) proposed three new parallel VNS procedures to deal with the *p*-median problem: synchronous parallel VNS (SP-VNS), replicated parallel VNS (RP-VNS) and replicated shaking VNS (RS-VNS). The main goal of the first method, SP-VNS, is to parallelize the local search strategy since, in general, it is the most time-consuming part. In SP-VNS, the neighbourhood is divided into n subsets, where n indicates the number of processors. Each subset is assigned to a different processor. Then, each processor returns an improved neighbour solution in its partition. Finally, the best neighbour is selected as the current solution to continue the search. The second approach, called RP-VNS, is directly a parallel independent multi-start method, which executes several independent VNS procedures. Specifically, RP-VNS is a multi-start procedure where each local search is replaced by a VNS. Each available processor, then, performs a sequential VNS. The parallel algorithm returns the best solution obtained by the processors. Finally, in the third parallel algorithm, RS-VNS, the shake and local search procedures are replicated as many times as the number of available processors. If the best local optimum found by the processors improves the previous solution (before executing the shake and improve methods), the RS-VNS procedure resorts to the first neighbourhood and continues the search. Otherwise, RS-VNS explores a larger neighbourhood. These three methods were tested on large instances derived from the travelling salesman problem library (TSPLIB) (Reinelt, 1991). In particular, the authors considered the instance RL1400 (with 1400 points) and derived nine different instances for the *p*-median problem where p ranges from 20 to 100. Reported results showed that the multi-start scheme (RP-VNS) obtained the best results.

Crainic *et al.* (2004) presented a different variant of parallel VNS, called cooperative neighbourhood VNS (CN-VNS). In this strategy, each individual VNS process (executed in a different processor) communicates exclusively with a central process called central memory or master. Therefore, there are no communications among individual VNS processes. The master is responsible for maintaining, updating and communicating the current overall best solution. The master also initiates and terminates the algorithm. Crainic *et al.* (2004) applied the CN-VNS method to the *p*-median problem, where each process (processor) implements the same VNS variant. The local search follows the first improvement strategy, implements fast interchange and considers $k_{\max} = p$, where $k_{\max}$ represents the maximum number of neighbourhoods that will be explored. Each processor, then, executes a 'normal' VNS exploration (shake, improve and neighbourhood change) while improving the current solution. When that solution is not improved, the corresponding processor communicates it to the master, requesting the best solution so far. The search is continued starting from the best overall solution in the current neighbourhood. The authors tested the CN-VNS method over a benchmark of problem instances from TSPLIB, where the number of customers ranges from 1400 to 11948 and the number of locations ranges from 10 to 1000. Reported results showed that the CN-VNS method reduces the CPU time without deteriorating the quality (when compared with the sequential VNS version). In addition, when both methods (parallel and sequential) are executed for the same CPU time, the parallel version finds better solutions.

Moreno *et al.* (2004) surveyed the aforementioned parallel VNS strategies. The authors analysed and tested them by considering large instances of the *p*-median problem. The paper concluded that cooperative mechanisms obtain, in general, better outcomes.

We propose in this paper six different variants of VNS which differ in how each one parallelizes the VNS stages. We group them into three different templates. 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 neighbourhoods. We illustrate the performance of these strategies by considering a specific VNS variant, called variable formulation search (VFS) (Pardo *et al.*, 2013), applied to the cutwidth minimization problem (CMP). It is important to remark that the

proposed methods are general parallelization strategies in the context of VNS, since they do not consider either the optimization problem or the VNS variant.

The rest of the paper is organized as follows. Section 2 describes the CMP and presents a brief survey of the main contributions to this problem. The sequential VNS method is briefly described in Section 3. Section 4 recalls the main parallel technologies, mainly focusing on Java threads. Section 5 presents the adaptation of the previous parallel VNS strategies as well as the new proposed methods. Section 6 presents and analyses the results of the computational experiments. Conclusions and perspectives are the subject of Section 7.

2. Cutwidth minimization problem

The CMP is an NP-Hard (Gavril, 1977) min–max layout problem. It consists of finding an ordering of the vertices of a graph on a line, in such a way that the maximum number of edges between each pair of consecutive vertices is minimized. The CMP has been formulated in both, combinatorial (Petit, 2003) and mathematical programming (Luttamaguzi *et al.*, 1977) ways. In this work, we consider the first of them. Given a graph $G = (V, E)$ with $|V| = n$, the CMP can be defined as follows. Let s be a labelling $s : V \rightarrow \{1, 2, \ldots, n\}$ of G that assigns the integers $\{1, 2, \ldots, n\}$ to the vertices in V, in such a way that each vertex receives a different label. The cutwidth of a vertex v with respect to s, denoted as $\mathrm{CW}_s(v)$, is the number of edges $(u, w) \in E$ satisfying $s(u) \leqslant s(v) < s(w)$, i.e.

$$\mathrm{CW}_s(v) = |\{(u, w) \in E : s(u) \leqslant s(v) < s(w)\}|.$$

The cutwidth of G with respect to s is defined as the maximum value of $\mathrm{CW}_s(v)$ for all $v \in V$. In mathematical terms, the objective function of the CMP is defined as

$$f(s) = \max_{v \in V} \mathrm{CW}_s(v).$$

The optimum cutwidth of G is then defined as the minimum $f(s)$ value over all possible labellings of G.

The CMP has also been referred to in the literature using alternative names such as Minimum Cut Linear Arrangement (Diaz *et al.*, 1997; Takagi & Takagi, 1999) or Network Migration Scheduling (Andrade & Resende, 2007a,b). There can be also found a generalization of the CMP for hypergraphs named Board Permutation Problem (Cohoon & Sahni, 1983, 1987). Several practical applications of this problem in different areas of Engineering and Computer Science, such as: Circuit Design (Adolphson & Hu, 1973; Cohoon & Sahni, 1987; Makedon & Sudborough, 1989), Network Reliability (Karger, 1999), Information Retrieval (Botafogo, 1993), Automatic Graph Drawing (Mutzel, 1995; Shahrokhi *et al.*, 2001), Protein Engineering or Networks Migration (Resende & Andrade, 2009) have also been reported. This last application refers to the problem where inter-nodal traffic from an obsolete telecommunications network needs to be migrated to a new network. This problem is now happening in phone traffic, where a migration between 4ESS switch-based networks to IP router-based networks (Resende & Andrade, 2009) is performed. Nodes are migrated, one at each time period, from the old to the new network. All traffic originating or terminating at a given node in the old network is moved to a specific node in the new network.

Let us consider this application in more detail as well as how it is related to the CMP. Suppose that the traffic from/to a node v_{old} in the old network is migrated to a node v_{new} in the new network. Let c_{old} be the capacity of the edge $(v_{\mathrm{old}}, v_{\mathrm{new}})$. The traffic between node v_{old} and node v_{new} must use a temporary link $(v_{\mathrm{old}}, v_{\mathrm{new}})$ connecting the two nodes with capacity c_{old}. When a node is migrated, one or more temporary links may need to be added, since v_{old} may be adjacent to more than one node still

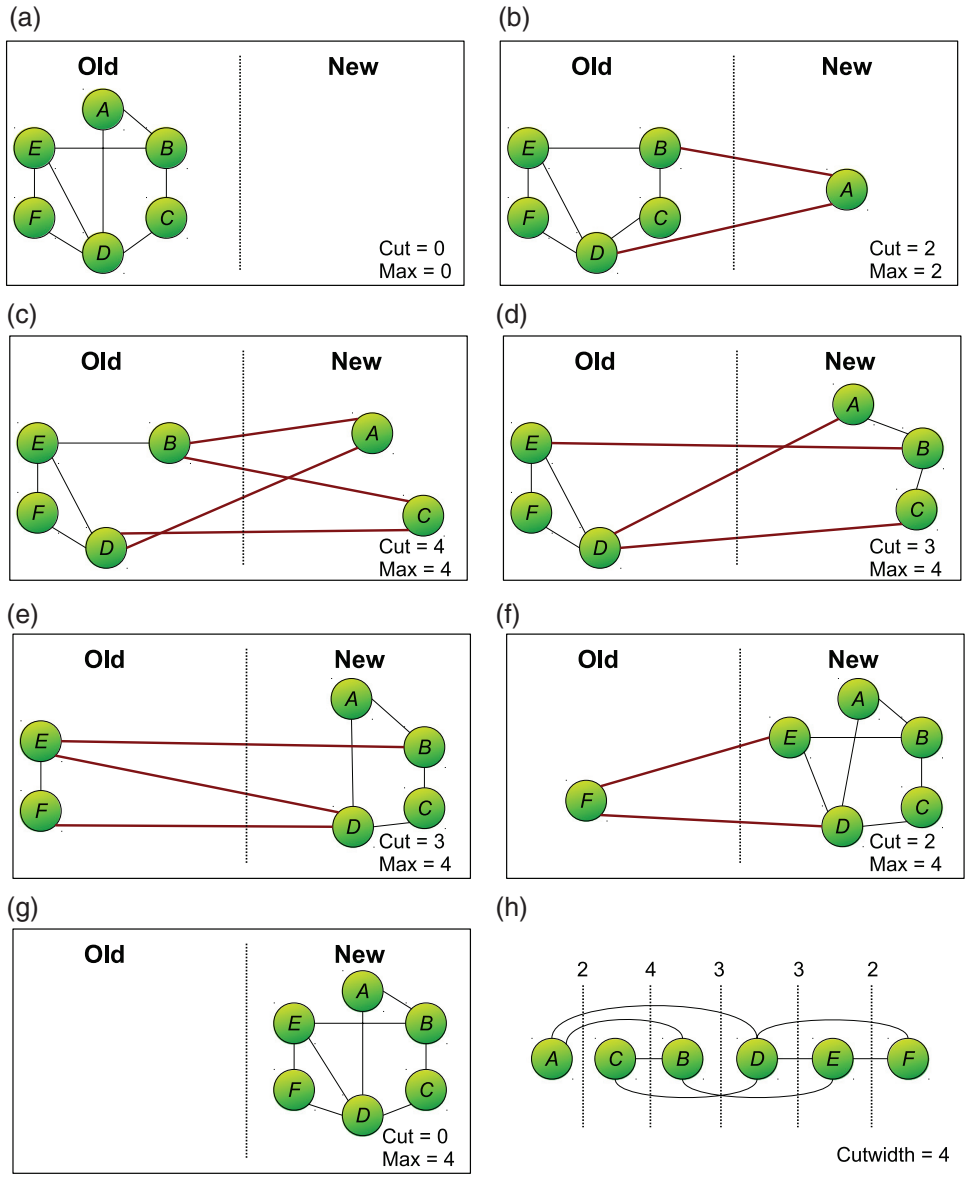

FIG. 1. (a)–(g) Node decommissioning process in a Network Migration Scheduling example and (h) the representation of the whole process as a linear layout.

active in the old network. A temporary link remains active until both nodes connected by the links are migrated to the new network.

The network is usually modelled as a graph $G = (V, E)$, where V is the vertex set, with $|V| = n$, and E is the edge set, with $|E| = m$. A solution to the Network Migration Scheduling Problem is an ordering (labelling, permutation, layout, etc.) of the nodes that need to be migrated. The CMP tries

to find the ordering that minimizes the maximum sum of capacities of the temporary links (edges). Figure 1(a–g) iteratively show the migration of a network. In particular, in Fig. 1(a) all the vertices are in the old network. Figure 1(b) shows the migration of the first vertex. In this case, we need to consider the inclusion of two temporary edges. Figure 1(c) shows the migration of the second vertex, which increases the maximum number of temporary edges to 4. Figure 1(g) shows the situation where all vertices have been migrated. Finally, Fig. 1(h) shows a solution of the CMP where the cut between each pair of edges represents the situation illustrated in Fig. 1(a–h). Note that the solution depicted in Fig. 1(h) is just a possible solution, but not the optimum one, since it is possible to find a different ordering (e.g. A, B, C, D, E, F) with a cutwidth value equal to 3.

In the scientific literature, the CMP has been addressed from both exact and heuristic approaches. Most of the exact approaches present polynomial-time algorithms for particular graphs, such as hypercubes (Harper, 1966), trees (Chung *et al.*, 1982; Yannakakis, 1985; Thilikos *et al.*, 2005) or grids (Rolim *et al.*, 1995). However, little work has been devoted to devise exact methods for general graphs. As far as the authors know, there are an Integer Linear Programming formulation (Luttamaguzi *et al.*, 1977) and two Branch and Bound algorithms (Palubeckis & Rubliauskas, 2012; Martí *et al.*, 2013) proposed in the literature for the CMP. These approaches can only solve relatively small instances. Therefore, different heuristic procedures have been proposed to address this problem. The work by Cohoon & Sahni (1987) was the first one in proposing heuristic algorithms for the generalized version of the problem. These authors proposed several constructive and local search procedures and embedded them in a Simulated Annealing MH. Twenty years later, Andrade & Resende (2007a,b) proposed a GRASP procedure hybridized with a Path Relinking method. Pantrigo *et al.* (2012) introduced a Scatter Search algorithm for the problem which outperformed previous results. The most recent approach (Pardo *et al.*, 2013) presented a new variant of the VNS schema, called VFS. This new algorithm is specially useful to deal with min–max (or max–min) problems, where it is usual that many solutions of the problem have associated the same value of the objective function. VFS makes use of alternative formulations of the problem to determine which solution is more promising when they have the same value of the objective function in the original formulation. The obtained results in the CMP show that the latter proposal outperforms the state-of-the-art algorithms in terms of quality and computing time. We propose in this paper six different strategies (three of them are adaptations of previous strategies to the CMP) to parallelize the VFS. The next section details the strategies to be parallelized in this work.

3. Variable formulation search

VFS is a new variant of the VNS MH proposed in Pardo *et al.* (2013). We refer the reader to this reference for a comprehensive description of the method. VFS tries to avoid getting trapped in a local optimum by considering different formulations of the optimization problem. This idea was originally introduced in Mladenović *et al.* (2005) in the context of the circle packing problem. The method changes formulations sequentially within the shaking, local search and neighbourhood change steps of the basic VNS. The aim is to determine whether a given solution is more promising than the other to continue the search, beyond the value of the original objective function. VFS is especially helpful in tackling problems which present a flat landscape where many solutions have associated the same value of the objective function. See Resende *et al.* (2010) and Duarte *et al.* (2011) for examples of other problems which present a flat landscape. The rest of this section is devoted to describing the method to construct the initial solution, the shake, local search and neighbourhood change procedures.

3.1 *Initial solution*

The initial solution is constructed with a GRASP procedure. In particular, this method labels the vertices sequentially (i.e. from 1 to n). To select the next vertex to be labelled, a candidate list (CL) is formed with all the unlabelled vertices that are adjacent to one or more vertices already labelled. Note that, in the first iteration, all vertices are candidates. For each vertex in the CL, an evaluation of its cutwidth is performed, considering that it is labelled with the next available label. Then, a restricted CL (RCL) is constructed with the vertices in the CL with a cutwidth value lower than a given threshold. Finally, a vertex is selected at random from the RCL and assigned the corresponding label. We used this procedure to construct a predefined number of solutions, selecting the best one as the initial solution for the VFS.

3.2 *Shake*

The shake strategy consists of performing perturbations in the current solution to diversify the search. Therefore, the local search (within VFS) starts the search from a new point with a different neighbourhood. The shake procedure receives a solution s and a parameter k representing the size of the perturbation. In particular, this procedure performs k interchange moves in s. Given a solution s and two different vertices $u, v \in V$, an interchange move produces a new solution s' where $s'(u) = s(v)$, $s'(v) = s(u)$ and $s'(w) = s(w)$ for all $w \in V$ not equal to u or v. Finally, the procedure finishes when k moves are performed, returning the corresponding perturbed solution.

3.3 *Local search*

The local search procedure allows one to reach a local optimum from the current (perturbed) solution at each iteration of the VFS. We designed a local search procedure based on insertion moves. Given a solution s, a vertex v placed in the position $s(v)$ and a position j such that $s(v) \neq j$, an insertion move consists of removing v from its current position $s(v)$ and inserting it in position j. This operation results in the ordering s', as follows:

- Let v_j be the vertex in position j in the ordering s. Then, if $s(v) = i$ is larger than j, then v is inserted just before v_j in position j. For example, considering the solution s

$$s = (\ldots, v_{(j-1)}, v_j, v_{(j+1)}, \ldots, v_{(i-1)}, v, v_{(i+1)}, \ldots),$$

  we would obtain the solution s':

$$s' = (\ldots, v_{(j-1)}, v, v_j, v_{(j+1)}, \ldots, v_{(i-1)}, v_{(i+1)}, \ldots).$$

- If $s(v) = i$ is smaller than j, v is inserted just after v_j in position j. Therefore, from the solution s:

$$s = (\ldots, v_{(i-1)}, v, v_{(i+1)}, \ldots, v_{(j-1)}, v_j, v_{(j+1)}, \ldots),$$

  we would obtain the solution s':

$$s' = (\ldots, v_{(i-1)}, v_{(i+1)}, \ldots, v_{(j-1)}, v_j, v, v_{(j+1)}, \ldots).$$

A naive idea using this kind of moves would be to try the insertion of any vertex in every position of the ordering. Therefore, the associated neighbourhood of this kind of moves has a relatively large size. However, taking into account the characteristics of the considered problem, it is possible to identify a set

of preferred positions for each vertex in the permutation. We refer the reader to Pardo *et al.* (2013) for an exhaustive description and formal derivation of these properties. As a consequence, the complexity of the local search is reduced from $O(n^2)$ to $O(n)$.

The resulting local search receives an initial solution, s, as an input parameter. Then, the procedure identifies the list C of vertices able to produce improvements if they are inserted in a different position of the ordering. The list C is then sorted according to the cutwidth of each vertex, in such a way that those vertices with a higher cutwidth value are evaluated earlier. The evaluation for each vertex v in C is then performed taking into account the aforementioned properties. If the move is accepted, the best solution found will be updated. Finally, the local search procedure ends when none of the moves performed with the vertices in the list C is able to produce an improvement, returning the best solution found.

3.4 *Neighbourhood change*

This procedure examines whether the solution obtained after the local search is better than any other solution previously found, or not. If so, the best solution found is updated and k (the value that determines which neighbourhood is explored) is set again to its initial value. Otherwise, k is increased with a constant value until the parameter $k_{\max}$ is reached, when a whole iteration of the VFS ends. As previously mentioned, the CMP presents a flat landscape and so, when two solutions have the same objective function value, it is hard to determine which one is better to carry on the search. For that reason, the VFS schema uses alternative formulations to compare two solutions, when it is not possible to determine which is the most promising one. In particular, the method first considers the original formulation of the problem. When two solutions present the same value of the objective function, the method uses an alternative formulation. Note that two formulations are equivalent when an optimum solution in one of them is also optimum in the other, although the value of the alternative objective function may have a different value. This schema can be repeated as far as new formulations of the problem are available. We refer the reader to Pardo *et al.* (2013) for a detailed description of the definition and use of alternative formulations.

4. Parallel technologies

The traditional Flynn classification of parallel architectures is based on two criteria: the number of instruction streams and the number of data streams that define four different classes: single instruction stream, single data stream (SISD); single instruction stream, multiple data streams (SIMD); multiple instruction streams, single data stream (MISD) and multiple instruction streams, multiple data streams (MIMD). Among these four models, MIMD is considered the most general model of parallel architectures (Alba & Nebro, 2005). Therefore, we consider MIMD architecture. This model has two kinds of memory models: shared memory (the whole memory can be accessed by each process) and distributed memory (each process has its own memory). Shared memory is considered the natural extension of the sequential programming. In fact, its foundations were established in the late 1960s to early 1970s (Alba & Nebro, 2005), so they are well known. In addition, common computers with several cores (processors) use a shared-memory model. Consequently, the easiest way of dealing with parallelism is the shared-memory model under the MIMD architecture. In this context, there are several technologies that can be used to implement parallel algorithms: OpenMP and threads (Pthreads and Java threads).

4.1 *OpenMP*

OpenMP is a set of compiler directives and library routines that can be used to implement parallel algorithms. The programmer then adds these compiler directives to a sequential program in order to inform

the compiler which part of the code must be concurrently executed. It is also possible to establish (if needed) synchronization points. 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 one compiler directive, without modifying the original sequential code. Therefore, OpenMP is adequate if the algorithm follows a data parallel model. However, it can be difficult to use in task parallel applications (i.e. not all the processes execute the same code).

4.2 *Threads*

In programming languages, a thread is an independent flow of control inside a process. Although it has always been possible to write parallel programs using processes and other resources provided by the operating system, multi-threaded processes are themselves concurrent programs that bring a number of advantages over multiple processes. In particular, they provide faster context switching among threads and lower resource usage.

Inside this paradigm, Pthreads and Java threads are considered the most representative tools. 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 multi-threaded programs portable. Java threads are a version of Pthreads for Java programming language.

Java threads offer the advantages of portability inherent in Java programs. It also provides a multi-threaded programming model adapted to the object-oriented features of Java. In addition, Java threads can be easily used to tackle task parallel applications. Therefore, we select this technology to implement our parallel algorithms.

5. Parallel VNS

The application of the parallelism to an MH can and must allow reducing the computational time (obtaining similar results to the sequential version) or increasing the exploration in the search space (obtaining better results than the sequential version). Notwithstanding, designing parallel MHs involves a considerable complexity since doing it appropriately implies that the researches must have a solid background in both fields. In general, these two fields are generally populated by distinct and very specialized groups of people. However, the rapid development of technology in designing processors (multi-core processors or dedicated architectures) has made use of parallel computing more and more popular.

According to Crainic *et al.* (2004), parallel MH strategies may be classified into one of the three following categories:

- *Low-level parallelism*: This strategy aims mainly at speeding up computations by executing in parallel one or several computing-intensive tasks (for instance, the local search) within one iteration of the method. It is usually implemented following the master–slave computing model. In particular, the master process dispatches work to the other processors (the ‘slaves’), recuperates and fuses the results, and then continues the sequential algorithm. Variants of this approach may be defined according to the quantity of work assigned to slave processors.
- *Domain decomposition*: The partitioning reduces the size of the solution space, but the procedure need to be repeated with different partitions to allow the exploration of the complete solution space. A master–slave scheme is often chosen as the implementation mechanism. A master process performs the decomposition and slave processors concurrently execute the MH on the resulting

sub-problems. Then, the master collects the partial solutions and builds a complete one. Finally, it decides on a new partition and the search is restarted.

- *Multiple search*: Parallelism is obtained from multiple concurrent explorations of the solution space. Concurrent searches may or may not execute the same heuristic method, and may start from the same or from different initial solutions. They may communicate during the search or only at the end to identify the best overall solution. The latter strategies are known as independent search methods, while the former are often called cooperative multi-search methods. Communications may be performed synchronously or asynchronously and may be event-driven or executed at predetermined or dynamically decided moments.

In this paper, we adapt three previously proposed VNS parallel strategies for the CMP. We additionally propose three new parallel strategies for this optimization problem. In particular, the six parallel VNS strategies fall in the 'Multiple Search' category. We do not propose low-level parallelism strategies since the shake, neighbourhood change and local search strategies are extremely fast (Pardo *et al.*, 2013). In the same line, we do not propose a domain decomposition parallel strategy since the shake, local search and neighbourhood change procedures needs to consider the whole solution (Pardo *et al.*, 2013). Therefore, the proposed methods are general parallelization strategies in the context of VNS, since they do not consider either the optimization problem or the VNS variant.

5.1 *Replicated parallel VNS*

The RP-VNS strategy consists of executing several parallel VNS procedures. We present two different strategies, RP-VNS1 and RP-VNS2. In particular, considering RP-VNS1, Algorithms 1 and 2 show the master and slave pseudo-code, respectively. The master process (Algorithm 1) starts by creating a pool of different threads (step 2). Then, for each thread in the pool, the master process sends the start signal to each thread (steps 3–5). When a thread finishes its execution, the master process compares the returned solution with the best overall solution, updating it if necessary (step 6).

Algorithm 1 RP-VNS1: master process

1: **function** `RP-VNS1-Master`(k_{max}, t_{max})
2: $P = CreateThreadPool(NTHREADS)$
3: **for all** $p \in P$ **do**
4: $\quad s_p \leftarrow ExecuteThread(p, k_{max}, t_{max})$
5: **end for**
6: $s_{best} = \arg\min_{p \in P}\{f(s_p)\}$
7: **return** s_{best}
8: **end**

Each slave process (Algorithm 2) executes the same VNS variant. Parameters $k_{\max}$ and $t_{\max}$ are set by the master process. The algorithm constructs an initial solution s in step 3 and sets k to the smallest neighbourhood. The current solution, s, is then perturbed with the shake procedure (step 6) obtaining a new solution s' in the kth neighbourhood. This solution is improved (step 7), resulting in a local optimum s''. The neighbourhood change procedure (step 8) determines whether s'' improves upon s (updating k to 1 and s to s'') or not (increasing the value of k). These steps are repeated until k reaches $k_{\max}$. Steps 2–11 are repeated until t reaches $t_{\max}$. Finally, the procedure returns the best solution found during the search.

Algorithm 2 RP-VNS1: slave process

1: **function** `RP-VNS1-Slave`(p, k_{max}, t_{max})
2: **repeat**
3: $s \leftarrow CreateInitialSolution()$
4: $k \leftarrow 1$
5: **repeat**
6: $s' \leftarrow Shake(s, k)$
7: $s'' \leftarrow LocalSearch(s')$
8: $NeighborhoodChange(s, s'', k)$
9: **until** $k = k_{max}$
10: $t \leftarrow CpuTime()$
11: **until** $t > t_{max}$
12: **return** s
13: **end**

Algorithm 3 RP-VNS2: master process

1: **function** `RP-VNS2-Master`(k_{max}, t_{max})
2: $P = CreateThreadPool(NTHREADS)$
3: $s_{best} \leftarrow CreateInitialSolution()$
4: **repeat**
5: **for all** $p \in P$ **do**
6: $s_p \leftarrow ExecuteThread(p, s_{best}, k_{max})$
7: **end for**
8: $s_{best} = \arg\min_{p \in P}\{f(s_p)\}$
9: **until** $t = t_{max}$
10: **return** s_{best}
11: **end**

The RP-VNS2 considers the cooperation among threads. In particular, each thread receives the best overall solution to start the corresponding VNS procedure. When a thread finishes, it notifies the master process the best solution found. Then, the master sends the best overall solution.

Algorithm 3 shows the master process pseudo-code. In this case, the master process constructs the initial solution (step 3) and controls the allowed execution time (steps 4–9). Additionally, it sends the best solution to each slave process (step 6). Algorithm 4 shows the slave pseudo-code. Now, this process does not construct a solution (which is received as an input argument), and it finishes when k reaches $k_{\max}$.

5.2 *Replicated shake VNS*

The RS-VNS strategy consists of parallelizing the shake and local search procedures. We consider two different strategies, RS-VNS1 and RS-VNS2. Algorithm 5 reports the pseudo-code of the RS-VNS1 master process. The algorithm begins by creating the pool of threads and an initial solution (steps 2 and 3). As usual, the VNS search is started in the smallest neighbourhood (step 5). Then, the master process delegates the execution of the shake and local search procedures to each thread. When a thread

Algorithm 4 RP-VNS2: slave process

1: **function** `RP-VNS1-Slave`(p, s, k_{max})
2: $s \leftarrow CreateInitialSolution()$
3: $k \leftarrow 1$
4: **repeat**
5: $\quad s' \leftarrow Shake(s, k)$
6: $\quad s'' \leftarrow LocalSearch(s')$
7: $\quad NeighborhoodChange(s, s'', k)$
8: **until** $k = k_{max}$
9: **return** s
10: **end**

Algorithm 5 RS-VNS1: master process

1: **function** `RS-VNS1-Master`(k_{max}, t_{max})
2: $P = CreateThreadPool(NTHREADS)$
3: $s_{best} \leftarrow CreateInitialSolution()$
4: **repeat**
5: $\quad k \leftarrow 1$
6: $\quad$ **repeat**
7: $\quad\quad$ **for all** $p \in P$ **do**
8: $\quad\quad\quad s_p \leftarrow ExecuteThread(p, s_{best}, k_{max})$
9: $\quad\quad$ **end for**
10: $\quad\quad s_{local} = \arg\min_{p \in P}\{f(s_p)\}$
11: $\quad\quad NeighborhoodChange(s_{best}, s_{local}, k)$
12: $\quad$ **until** $k = k_{max}$
13: **until** $t = t_{max}$
14: **return** s_{best}
15: **end**

finishes, it returns the best found solution (step 8). Once all threads have finished, the master process gathers the best solution found by the threads (step 10). The neighbourhood change procedure (step 11) determines whether s'' improves upon s (updating k to 1 and s to s'') or not (increasing the value of k). These steps are repeated until k reaches $k_{\max}$. Steps 4–13 are repeated until t reaches $t_{\max}$. Finally, the master process returns the best solution found during the search. For the sake of brevity, we do not include a pseudo-code of the slave process since it only consists of two instructions (shake and local search functions).

This strategy is equivalent to a best improvement method in the sense that the search waits until all threads finish, performing the best available move. RS-VNS2 considers an alternative strategy. In particular, it performs the first move which improves the current best found solution (first improvement), instead of waiting for all threads. Specifically, the pseudo-code of RS-VNS2 is similar to RS-VNS1. For that reason, we do not include the associated pseudo-code. The only difference between them is which solution is used in the neighbourhood change procedure. In RS-VNS2, we compare the solution returned by each thread with the best found solution so far. In case of improvement, we update the corresponding

best solution and stop the execution of the remaining threads. Note that in RS-VNS2, step 10 must also be removed, since it is not required to find the best solution among all the treads.

5.3 *Cooperative neighbourhood VNS*

The CN-VNS strategy considers the cooperative exploration of different neighbourhoods by different threads. We present two different strategies: CN-VNS1 and CN-VNS2. In both, the master process is responsible for maintaining, updating and communicating the current overall best solution. It also initiates and terminates the algorithm executed in each thread. The logic of the master process is similar to the one presented in Algorithm 5. The main difference resides in the management of the neighbourhood. In particular, both strategies (CN-VNS1 and CN-VNS2) delegate this task to each thread. Therefore, the pseudo-code of these two strategies are equivalent to RS-VNS, but removing the instruction in step 11 (neighbourhood change procedure) and the input argument k in step 8 (since this value is set in the slave process).

In CN-VNS1, the slave process explores the $k_{\max}$ available neighbourhoods at random, while in CN-VNS2, the exploration is performed systematically. Pseudo-codes of CN-VNS1 and CN-VNS2 are shown in Algorithms 6 and 7, respectively. In particular, the random exploration of neighbourhoods starts by selecting at random a value of k, between 1 and $k_{\max}$ (step 3). Considering the solution s, the procedure performs the corresponding shake (step 5), obtaining a perturbed solution s' in the kth neighbourhood. This solution is then improved with the local search method (step 6), obtaining s''. Then, it is decided whether the slave process performs another iteration or not. Specifically, the shake and local search procedures are repeated while s'' is better than s (see steps 7–11); otherwise, the thread communicates the best solution found to the master process. Therefore, the corresponding thread waits until the master process sends again the best overall solution.

In CN-VNS2, slave processes explore systematically the $k_{\max}$ available neighbourhoods. In order to do so, the thread p performs the search in the $[k^p_{\text{first}}, k^p_{\text{last}}]$ sub-ranges. For example, the first thread explores neighbourhoods from 1 to $\lfloor k_{\max}/|P| \rfloor$, the second thread explores the neighbourhoods from $\lfloor k_{\max}/|P| \rfloor + 1$ to $\lfloor 2 * k_{\max}/|P| \rfloor$ and so on. Then, starting from solution s, the algorithm perturbs it (step 7), obtaining a new solution s' in the kth neighbourhood, with $k^p_{\text{first}} \leqslant k \leqslant k^p_{\text{last}}$. This solution is then

Algorithm 6 CN-VNS1: slave process

```
1:  function CN-VNS1-Slave(p, s, k_max)
2:  improved ← TRUE
3:  k ← Random(1, k_max)
4:  while improved = TRUE do
5:      s' ← Shake(s, k)
6:      s'' ← LocalSearch(s')
7:      if f(s'') > f(s) then
8:          improved ← FALSE
9:      else
10:         s ← s''
11:     end if
12: end while
13: return s
14: end
```

Algorithm 7 CN-VNS2: slave process

1: **function** `CN-VNS2-Slave`(p, s, k_{max})
2: $improved \leftarrow TRUE$
3: $k_{first} = \lfloor (p-1) * k_{max}/|P| \rfloor + 1$
4: $k_{last} = \lfloor p * k_{max}/|P| \rfloor$
5: $k \leftarrow k_{first}$
6: **repeat**
7: $s' \leftarrow Shake(s, k)$
8: $s'' \leftarrow LocalSearch(s')$
9: **if** $f(s'') < f(s)$ **then**
10: $s \leftarrow s''$
11: $k \leftarrow k_{first}$
12: **else**
13: $k \leftarrow k + 1$
14: **end if**
15: **until** $k = k_{last}$
16: **return** s
17: **end**

improved with the local search method (step 8), obtaining s''. Then, it is decided whether s'' improves upon s (resetting k to k_{first} and assigning s'' to s) or not (increasing the value of k). The thread finishes the search when k reaches k_{last}, communicating the best solution found to the master process. Then, it waits until the master process again sends it the best overall solution.

6. Computational experiments

This section reports and analyses the computational experiments that we have performed for testing the efficiency of the six proposed parallel VNS variants for solving the CMP. All the algorithms were implemented in Java SE 7 and the experiments were conducted on an Intel Core i7 2600 CPU (3.4 GHz) and 4 GB RAM. We derived 101 instances from the Harwell-Boeing Sparse Matrix Collection. This collection consists of a set of standard test matrices $M = M_{uv}$ arising from problems in linear systems, least squares and eigenvalue calculations from a wide variety of scientific and engineering disciplines. The graphs are derived from these matrices by considering an edge (u, v) for every element $M_{uv} = 0$. From the original set, we have selected the 101 graphs with $n \leqslant 3025$. The number of vertices and edges ranges from 30 to 3025 and from 103 to 8904, respectively.

We have divided our experimentation in two different parts: preliminary experimentation and final experimentation. In the preliminary experimentation, we study the effect of the number of threads in each algorithm. We consider a representative subset of 14 instances to conduct the preliminary experimentation. Then, in the final experimentation, we compare the best variants with the state-of-the-art algorithm.

According to Crainic & Toulouse (2003), the classical performance measure (i.e. speedup described by Barr & Hickman, 1993) is not adequate to evaluate the performance of parallel MHs since asynchronous interactions between threads generally induce significant differences in search behaviour, not only for the global parallel method, but also for each search process participating in the cooperation. Therefore, the sequential and parallel methods may then be viewed as different MHs, requiring a redefinition of speedup and other performance measures. This situation is further aggravated by the

TABLE 1 *Performance of RP-VNS*

	RP-VNS1				RP-VNS2			
Threads	2	4	8	16	2	4	8	16
Avg.	105.29	105.29	106.14	106.50	107.64	107.07	106.14	106.79
Dev (%)	2.03	2.03	3.48	4.41	7.77	6.93	7.11	7.44
#Best	9	9	8	8	7	7	8	7

randomness embedded in the VNS methods considered in this paper. However, it is important to remark that a parallelization strategy should speed up the search or 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.

To have an illustrative comparison, each parallel variant is executed for the same CPU time than the best method identified in the literature, i.e. the VFS introduced by Pardo *et al.* (2013). In particular, we execute VFS (considering the parameters suggested by the authors) over the set of 14 instances used in the preliminary experimentation. The average CPU time of VFS is 840 s. Therefore, we execute all parallel VNS variants for this computing time.

In our first experiment, we compare the performance of RP-VNS1 and RP-VNS2 (described in Section 5.1) considering different number of threads: 2, 4, 8 and 16. Table 1 reports the average quality over all instances (Avg.), the average percent deviation with respect to the best known solutions (Dev (%)) and number of times that each method matches the best known solutions (#Best). Let s be a heuristic solution whose objective function value is $f(s)$ and let $s^\star$ be the best-known solution (with objective function value $f(s^\star)$). The deviation is then computed as $\text{Dev} = (f(s) - f(s^\star))/f(s^\star)$. We consider these three performance metrics for the rest of the experiments since they complement each other.

Table 1 shows that RP-VNS1 clearly outperforms RP-VNS2 in all the considered statistics. This results can be partially explained by the fact that the diversification of RP-VNS1 is larger than PR-VNS2 (i.e. it explores a larger number of different regions of the search space), since the last method starts the search from the best-known solution found so far. As it is well documented in the literature, the CMP presents a flat landscape. As a consequence, most of the moves performed by the search procedure have associated a null value. Therefore, it is more interesting to start the search from other point of the solution space (RP-VNS1) rather than continuing the search from the same point (RP-VNS2). We observe that better outcomes are obtained with a lower number of threads. As a result, RP-VNS1 with four threads is selected as the best variant of RP-VNS. We will use this algorithm in the final experimentation.

In the second experiment, we compare the performance of RS-VNS1 and RS-VNS2 (described in Section 5.2). As in the previous experiment, we consider that the number of threads ranges from 2 to 16. Table 2 reports that differences between both variants are smaller than in the first experiment. In particular, RS-VNS1 with 4 threads presents an average percentage deviation of 1.38%, and it matches 11 times the best-known solutions (out of 14). The best RS-VNS2 method uses 16 threads and finds the same number of best values. However, it presents a slightly larger deviation (1.45%). Therefore, we select RS-VNS1 with four threads as RS-VNS variant for the final experimentation.

TABLE 2 *Performance of RS-VNS*

	RS-VNS1				RS-VNS2			
Threads	2	4	8	16	2	4	8	16
Avg.	106.50	105.14	105.43	105.36	105.79	105.36	105.07	105.07
Dev (%)	3.29	1.38	2.37	2.15	3.50	2.35	1.45	1.45
#Best	7	11	8	10	8	8	10	11

TABLE 3 *Performance of CN-VNS*

	CN-VNS1				CN-VNS2			
Threads	2	4	8	16	2	4	8	16
Avg.	110.29	108.64	108.64	108.50	107.64	108.43	107.57	107.50
Dev (%)	10.25	9.59	9.56	8.82	7.53	7.83	7.51	7.24
#Best	8	7	7	7	7	7	7	7

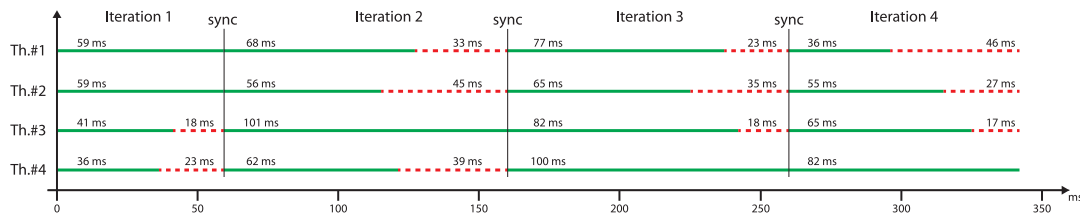

FIG. 2. Performance of each thread in RS_VNS1 on *lshp3025* instance.

In the next preliminary experiment, we compare the performance of two CN-VNS (i.e. CN-VNS1 and CN-VNS2, both described in Section 5.3). Table 3 reports the results obtained by both parallel algorithms, considering the same number of threads than in the previous experiments. The performance of all these variants seems to be really poor compared with the previous methods. In particular, CN-VNS2 with 16 threads is the best variant with a deviation of 7.24% and matches the best known solution 7 times (out of 14). It seems that the larger the number of threads, the better is the performance of the method. Then this strategy could be more competitive if we would consider a larger number of threads. However, due to our hardware limitation, we are not able to test this hypothesis. Specifically, if we increase the number of threads, the required time to manage them deteriorates the performance of the method. It is interesting to compare the best CN-VNS variant with the state-of-the-art method.

In the last two preliminary experiments, we study the time spent in the parallelization management of the algorithms. First, in Fig. 2, we illustrate the behaviour of the different threads within an algorithm. In order to do that, we have selected RS-VNS1 with four threads and we have depicted its behaviour over a representative instance (*lshp3025*) during four iterations. For each iteration and thread, we depict the time that the thread is working (straight lines) and the time that the thread is waiting for synchronization (dashed lines). As shown in the figure, an iteration ends when all threads have finished working. After that, the parallel algorithm synchronizes the information among the threads (represented in the figure as

TABLE 4 *Comparison of the synchronization time spent by each algorithm*

	RP-VNS1 (4) (%)	RS-VNS1 (4) (%)	CN-VNS2 (16) (%)
Avg. waiting time	9.43	1.44	0.04
Max. waiting time	11.25	4.13	0.07

vertical black lines at the end of each iteration). Obviously, there is at least one thread that is working until the end of the iteration.

The total computing time of the previous example is 342 ms, which includes the working time, the waiting time and the synchronization time of each thread. In all our experiments, the synchronization time is negligible; therefore, we do not consider further analysis of this time in the performance of the parallel strategies.

In the example shown in Fig. 2, we can observe that the thread that waits longer (Th.#2) spends approximately 31% of the time waiting, while Th.#3 is the one with the minimum waiting time (approximately 15% of the time). On average, the threads are waiting 23% of the time.

Once the behaviour of the threads has been described, we conduct an additional experiment over the set of instances used in the preliminary experimentation. In Table 4, we present the best variants of the three proposed parallel algorithms (where the number of threads are depicted in parenthesis). For each method, we report the average CPU time spent by all threads waiting for synchronization (Avg. waiting time) and also the maximum CPU time that a single thread within the method needs to wait (Max. waiting time). Both results are presented in percentage over the total CPU time spent. As expected, RP-VNS1 presents the largest average and maximum waiting time, since each thread in this strategy performs a completely independent iteration. In other words, if the constructed solution has a relatively high quality, the VNS strategy will reach a local optimum faster. Therefore, the waiting time of that thread will be longer, since it will need the other threads (starting from a worse solution) to finish. On the other hand, the CN-VNS2 strategy presents the shortest average and maximum waiting time. This behaviour can be explained by considering that, in this strategy, all the threads start the search from the same initial solution (the best one found in the iteration). Therefore, the iterations performed by the parallel VNS are quite similar. Finally, the RS-VNS1 strategy meets a balance among the strategies of the other two methods. Particularly, at each iteration it starts from the best solution found (similar to CN-VNS2) but it applies an independent shake and local search procedures to that solution in each thread (similar to RP-VNS1).

Finally, we compare the previous best variants with the state-of-the-art method (VFS). In this experiment, we consider the whole set of 101 instances. Table 5 clearly shows the superiority of the RS-VNS strategy over the other competitors. In particular, it presents a deviation of 0.28% which compares favourably to the RP-VNS (0.64%), the state-of-the-art method (0.89%) and the CN-VNS (1.52%). Considering the number of best solutions found, the analysis is similar. It is important to remark that, although CN-VNS2 (16) obtains the worst deviation, it only finds one less better solution than the state-of-the-art method. Even more, when considering the average quality, the CN-VNS2 (16) ranks second (284.87), improving the RP-VNS1 (4) with an average quality of 285.01 and VFS (285.46).

To confirm these conclusions, we conduct the non-parametric Friedman test (Friedman, 1940) for multiple correlated samples to the best solutions obtained by VFS, RP-VNS1, RS-VNS1 and CN-VNS2. This test computes, for each instance, the rank value of each method according to the solution quality (where rank 1 is assigned to the best method and rank 4 to the worst one). Then it calculates the average rank values of each method across all the instances solved. If the averages differ greatly, the associated

TABLE 5 *Comparison of the parallel VNS variants with the state of the art*

	VFS	RP-VNS1(4)	RS-VNS1(4)	CN-VNS2(16)
Avg.	285.46	285.01	284.64	284.87
Dev (%)	0.89	0.64	0.28	1.52
#Best	84	86	92	83

p-value or significance will be small. The resulting *p*-value of 0.028 (considering a level of significance of 0.05) obtained in this experiment clearly indicates that there are statistically significant differences among the four methods tested. Specifically, the rank values produced by this test are 2.20 (RS-VNS1), 2.45 (RP-VNS1), 2.47 (VFS) and 2.88 (CN-VNS2). This experiment confirms again the superiority of two parallel VNS procedures over the state of the art.

We conduct a test of Wilcoxon to further analyse the differences between our best method RS-VNS1 and the state-of-the-art algorithm (VFS). In particular, this experiment has a *p*-value of 0.024. Then, considering a level of significance of 0.05, this experiment indicates that there are statistically significant differences between both methods, confirming again that the parallel version improves the sequential one.

7. Conclusion

We proposed in this paper six different general parallel designs for the VNS schema. We group these variants into three different templates. The first one is oriented to parallelize the whole VNS method (RP-VNS1 and RP-VNS2). The second one parallelizes the shake and the local search procedures (RS-VNS1 and RS-VNS2). Finally, the third one explores in parallel the set of predefined neighbourhoods (CN-VNS1 and CN-VNS2). These general strategies are then applied to parallelize a recently introduced VNS algorithm, called VFS. We conducted an experimental comparison among these variants on the CMP. A preliminary experimentation allows us to identify the best variant from each template, resulting in the selection of RP-VNS1, RS-VNS1 and CN-VNS2, respectively. Then we compared the selected three methods with the sequential VFS, which is the best algorithm in the state of the art for the CMP. Experimental results showed that two of the three proposed parallel methods (RS-VNS1 and RP-VNS1) outperform previous best methods in the state of the art of the CMP in terms of quality. We also conducted statistical tests to confirm the significance of the obtained results with RS-VNS1 emerging as the best algorithm.

Funding

This research was partially supported by the Spanish Ministry of 'Economía y Competitividad', grants ref. TIN2009-07516, TIN2011-28151 and TIN2012-35632, and the Government of the Community of Madrid, grant ref. S2009/TIC-1542.

References

ADOLPHSON, D. & HU, T. C. (1973) Optimal linear ordering. *SIAM J. Appl. Math.*, **25**, 403–423.

ALBA, E. & NEBRO, A. J. (2005) New technologies in parallelism. *Parallel Metaheuristics. A New Class of Algorithms*, vol. 2. Wiley-Interscience.

ANDRADE, D. V. & RESENDE, M. G. C. (2007a) GRASP with path-relinking for Network Migration Scheduling. *Proceedings of International Network Optimization Conference (INOC).*

Andrade, D. V. & Resende, M. G. C. (2007b) GRASP with evolutionary path-relinking. *Proceedings of Seventh Metaheuristics International Conference (MIC)*.

Barr, R. S. & Hickman, B. L. (1993) Reporting computational experiments with parallel algorithms: issues, measures, and experts opinions. *ORSA J. Comput.*, **5**, 2–18.

Botafogo, R. A. (1993) Cluster analysis for hypertext systems. *16th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pp. 116–125.

Chung, M. J., Makedon, F., Sudborough, I. H. & Turner, J. (1982) Polynomial time algorithms for the MIN CUT problem on degree restricted trees. *SFCS '82: Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, pp. 262–271.

Crainic, T. G., Gendreau, M., Hansen, P. & Mladenović, N. (2004) Cooperative parallel variable neighborhood search for the p-median. *J. Heuristics*, **10**, 293–314.

Crainic, T. G. & Toulouse, M. (2003) Parallel strategies for meta-heuristics. *State-of-the-Art Handbook in Metaheuristics*, pp. 475–513.

Cohoon, J. & Sahni, S. (1983) Exact algorithms for special cases of the Board Permutation Problem. *Proceedings of the Allerton Conference on Communication, Control and Computing*, pp. 246–255.

Cohoon, J. & Sahni, S. (1987) Heuristics for backplane ordering. *J. VLSI Comput. Syst.*, **2**, 37–61.

Díaz, J., Gibbons, A., Pantziou, G. E., Serna, M. J., Spirakis, P. G. & Toran, J. (1997) Parallel algorithms for the minimum cut and the minimum length tree layout problems. *Theor. Comput. Sci.*, **181**, 267–287.

Duarte, A., Martí, R., Resende, M. G. C. & Silva, R. M. A. (2011) GRASP with path relinking heuristics for the antibandwidth problem. *Networks*, **58**, 171–189.

Friedman, M. (1940) A comparison of alternative tests of significance for the problem of m rankings. *Ann. Math. Statist.*, **11**, 86–92.

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

Gavril, F. (1977) Some NP-complete problems on graphs. *Proceedings of the Eleventh Conference on Information Sciences and Systems*, pp. 91–95.

Gendreau, M. & Potvin, J. Y. (1977) *Handbook of Metaheuristics*. Berlin: Springer Publishing Company, Incorporated.

Harper, L. H. (1966) Optimal numberings and isoperimetric problems on graphs. *J. Combin. Theory*, **1**, 385–393.

Karger, D. R. (1999) A randomized fully Polynomial Time Approximation Scheme for the all-terminal network reliability problem. *SIAM J. Comput.*, **29**, 492–514.

Luttamaguzi, J., Pelsmajer, M., Shen, Z. & Yang, B. (2005) Integer programming solutions for several optimization problems in graph theory. *Technical Report, Center for Discrete Mathematics and Theoretical Computer Science, DIMACS*.

Makedon, F. & Sudborough, I. H. (1989) On minimizing width in linear layouts. *Discrete Appl. Math.*, **23**, 243–265.

Martí, R., Pantrigo, J. J., Duarte A. & Pardo, E. G. (2013) Branch and bound for the cutwidth minimization problem. *Comput. Oper. Res.*, **40**, 137–149.

Mladenović, N. & Hansen, P. (1997) Variable neighborhood search. *Comput. Oper. Res.*, **24**, 1097–1100.

Mladenović, N., Plastria, F. & Urosevic, D. (2005) Reformulation descent applied to circle packing problems. *Comput. Oper. Res.*, **32**, 2419–2434.

Moreno, J. A., Moreno, J. M. & Verdegay, J. L. (2004) Parallel Variable neighborhood search. The *p*-median problem: a survey of metaheuristic approaches. *Les Cahiers du GERAD*, **92**, 1–22.

Mutzel, P. (1995) A polyhedral approach to planar augmentation and related problems. *ESA'95: Proceedings of the Third Annual European Symposium on Algorithms*, pp. 494–507.

Palubeckis, G. & Rubliauskas, D. (2012) A branch-and-bound algorithm for the minimum cut linear arrangement problem. *J. Combin. Optim.*, **24**, 540–563.

Pantrigo, J. J., Martí, R., Duarte, A. & Pardo, E. G. (2012) Scatter search for the cutwidth minimization problem. *Ann. Oper. Res.*, **199**, 285–304.

Pardo, E. G., Mladenovic, N., Pantrigo, J. J. & Duarte, A. (2013) Variable formulation search for the cutwidth minimization problem. *Appl. Soft Comput.*, **13**, 2242–2252.

Petit, J. (2003) Experiments on the minimum linear arrangement problem. *ACM J. Exp. Algorithmics*, **8**, 1084–6654.

Reinelt, G. (1991) TSPLIB—a traveling salesman problem library. *INFORMS J. Comput.*, **3**, 376–384.

Resende, M. G. C. & Andrade, D. V. (2009) *Method and System for Network Migration Scheduling*. Atlanta, Georgia: United States Patent Application Publication, US2009/0168665.

Resende, M. G. C., Martí, R., Gallego, M. & Duarte, A. (2010) GRASP and path relinking for the max-min diversity problem. *Comput. Oper. Res.*, **37**, 498–508.

Rolim, J., Skora, O. & Vrt'o, I. (1995) Optimal cutwidths and bisection widths of 2- and 3-dimensional meshes. *Graph-Theoretic Concepts in Computer Science*, vol. 1017, pp. 252–264.

Shahrokhi, F., Skora, O., Szkely, L. A. & Vrt'o, I. (2001) On bipartite drawings and the linear arrangement problem. *SIAM J. Comput.*, **30**, 1773–1789.

Takagi, K. & Takagi, N. (1999) Minimum Cut Linear Arrangement of p-q dags for VLSI layout of adder trees. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, **E82-A**, 767–774.

Thilikos, D. M., Serna, M. J. & Bodlaender, H. L. (2005) Cutwidth II: algorithms for partial w-trees of bounded degree. *J. Algorithms*, **56**, 25–49.

Yannakakis, M. (1985) A polynomial algorithm for the Min-Cut linear arrangement of trees. *J. ACM*, **32**, 950–988.