Computers & Operations Research 52 (2014) 209–219

Contents lists available at ScienceDirect

Computers & Operations Research

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

Combining intensification and diversification strategies in VNS. An application to the Vertex Separation problem

Jesús Sánchez-Oro, Juan José Pantrigo, Abraham Duarte *

Dpto. de Ciencias de la Computación, Universidad Rey Juan Carlos, Móstoles, Madrid, Spain

ARTICLE INFO

Available online 18 November 2013

Keywords:
Combinatorial optimization
Intensification
Diversification
Variable Neighborhood Search
Layout problems
Vertex separation problem

ABSTRACT

The Vertex Separation problem (VSP) is an NP-hard problem with practical applications in VLSI design, graph drawing and computer language compiler design. VSP belongs to a family of optimization problems in which the objective is to find the best separator of vertices or edges in a generic graph. In this paper, we propose different heuristic methods and embed them into a Variable Neighborhood Search scheme to solve this problem. More precisely, we propose (i) a constructive algorithm, (ii) four shake procedures, (iii) two neighborhood structures, (iv) efficient algorithmic strategies to explore them, (v) an extended version of the objective function to facilitate the search process and finally, (vi) we embed these strategies in a Reduced Variable Neighborhood Search (RVNS), a Variable Neighborhood Descent (VND) and a General Variable Neighborhood Search (GVNS). Additionally, we provide an extensive experimental comparison among them and with the best previous method of the literature. We consider three different benchmarks, totalizing 162 representative instances. The experimentation reveals that our best procedure (GVNS) improves the state of the art in both quality and computing time. This fact is confirmed by non-parametric statistical tests. In addition, when considering only the largest instances, the other two proposed variants (RVNS and VND) also obtain statistically significant differences with respect to the best previous method identified in the state of the art.

1. Introduction

Graph layout problems are a class of combinatorial optimization problems where the objective consists of finding a linear labeling of the vertices of a graph in such a way that a specific objective function is maximized or minimized. Given an undirected and unweighted graph $G(V,E)$ where V (with $|V|=n$) and E (with $|E|=m$) are the sets of vertices and edges, respectively, a linear labeling φ of the vertices of G is a bijection $\varphi: V \rightarrow \{1,2,\ldots,n\}$ which assigns a unique and different integer between 1 and n to each vertex $u \in V$. A labeling has also been named as linear ordering, linear arrangement layout, numbering or simply, ordering. A common way to represent a layout problem is to dispose the vertices of a graph in a straight line, where each vertex u is allocated in position $\varphi(u)$. Fig. 1a shows an example of a graph, G, with 7 vertices and 11 edges, and Fig. 1b an example of a layout φ.

The Linear Arrangement [27], Bandwidth [25], Cutwidth [23] or Antibandwidth [7] fall into this class of optimization problems. In this paper, we tackle the Vertex Separation problem (VSP). Before defining mathematically the VSP, we need to introduce some definitions. Let $L(p,\varphi,G)$ be the set of vertices in V with a position in the layout φ lower than or equal to position p. Symmetrically, let $R(p,\varphi,G)$ be the set of vertices with a position in the layout φ larger than position p. In mathematical terms,

$$L(p,\varphi,G) = \{v \in V : \varphi(v) \leq p\},$$

$$R(p,\varphi,G) = \{v \in V : \varphi(v) > p\}.$$

In general, $L(p,\varphi,G)$ can be simply referred to as the set of *left vertices* with respect to position p in φ. Symmetrically, $R(p,\varphi,G)$ is the set of the *right vertices* with respect to p in φ. We define the separation value at position p of layout φ, $Sep(p,\varphi,G)$, as the number of vertices in $L(p,\varphi,G)$ with one or more adjacent vertices in $R(p,\varphi,G)$. More formally

$$Sep(p,\varphi,G) = |\{u \in L(p,\varphi,G) : \exists v \in R(p,\varphi,G) \wedge (u,v) \in E\}|.$$

The Vertex Separation value (VS) of layout φ is the maximum of the *Sep*-value among all positions in φ:

$$VS(\varphi,G) = \max_{1 \leq p < n} Sep(p,\varphi,G).$$

The Vertex Separation problem (VSP) then consists of finding a layout, $\varphi^\star$, minimizing the *VS*-value of the graph G. In mathematical

* Corresponding author. Tel.: +34 914888116.
E-mail address: abraham.duarte@urjc.es (A. Duarte).

0305-0548/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.cor.2013.11.008

terms

$$\varphi^{\star} = \arg\min_{\varphi \in \Phi} VS(\varphi, G),$$

where Φ represents the set of all possible layouts of G.

Fig. 2a–f depicts the *Sep*-value of each position p of layout φ, $Sep(p, \varphi, G)$. For instance, $Sep(1, \varphi, G) = 1$ (see Fig. 2a) because $L(1, \varphi, G) = \{C\}$ and $R(1, \varphi, G) = \{A, D, E, B, F, G\}$ and there is only one vertex in L having an adjacent vertex in R. Similarly, $Sep(5, \varphi, G) = 4$ (2e) because $L(5, \varphi, G) = \{C, A, D, E, B\}$ and $R(3, \varphi, G) = \{F, G\}$ and there are four vertices $\{C, A, D, B\}$ in L that has an adjacent vertex in R. Notice that vertex E is not highlighted in Fig. 2e, since it has no adjacent vertices in R. Therefore, it does not contribute to the *Sep*-value. The objective function is then computed as the maximum of these *Sep*-values. In particular, for the example depicted in Fig. 2, this value is $VS(G, \varphi) = 4$, associated to position $p = 5$.

The VSP is strongly related to other well-known graph problems, such as the Path-width [17], the Node Search Number [19] or the Interval Thickness [18], among others. The description of the equivalences among these problems can be found in [12,17,19]. All these optimization problems are NP-hard and have practical applications in VLSI design [20], computer language compiler design [4] or graph drawing [9].

We can find efficient exact approaches to solve the VSP on special classes of graphs. A linear algorithm to compute the optimal Vertex Separation of a tree is proposed in [10] as well as a $O(n \log n)$ algorithm for finding the corresponding optimal layout. The algorithm was improved in [28] with a linear time procedure to find the optimal layout. In [24] an alternative method to compute the Vertex Separation of trees was proposed. In [11] a $O(n \log n)$ algorithm to compute the Vertex Separation of unicyclic graphs (i.e., trees with an extra edge) is proposed. A polynomial-time algorithm to compute the Path-Width (which is identical to VSP) is proposed in [2]. However, the algorithm cannot be considered from a practical point of view, since the bound on its time complexity is $\Omega(n)$, see [11]. In [5] it is proposed a polynomial time algorithm for optimally solving the VSP for n-dimensional grids. Co-graphs and permutational graphs can also be optimally solved as it was proposed in [1,2], respectively. Approximation algorithms have also been proposed for the VSP. Specifically, [3] proposes a polynomial time $O(\log^2 n)-$approximation algorithm for general graphs and a $O(\log n)-$approximation algorithm for planar graphs. Similar results for binomial random graphs are presented in [6]. Finally, [8] proposed a Basic VNS for the VSP. In particular, the authors presented two constructive procedures and one local search based on interchange moves. As far as we know, this algorithm obtains the best results in this problem, so we use it to test the performance of our proposals.

The main objective of this paper is to provide experimental evidences that, in the context of VNS, the compromise between intensification and diversification usually obtains the best results. In order to do so, we propose three VNS algorithms (RVNS, VND, and GVNS), where RVNS mainly focuses on the diversification, VND mainly focuses on the intensification, and GVNS balances intensification and diversification. In this line, we also propose new strategies to combine intensification and diversification within shake procedures. Computational experiments reveal that our best procedure (GVNS) improves the state of the art in both quality and computing time. This fact is confirmed with non-parametric statistical tests. In addition, when considering only the largest instances, the other two proposed variants (RVNS and VND) also obtain statistically significant differences with respect to the best previous method identified in the state of the art.

The rest of the paper is organized as follows. Section 2 introduces the VNS variants studied in this paper. We propose a new constructive procedure for the VSP (see Section 3.1). We provide a formal definition of two new neighborhood structures for the VSP based on insert moves (see Section 3.2). We then introduce four different shake procedures with different balance between diversification and intensification (see Section 3.3). In Section 3.4 we propose an extended version of the objective function which provides a more convenient view about the quality of the solutions. Additionally, we include an efficient strategy to perform insert moves (see Section 3.5) and algorithmic methods to explore the neighborhood structures (Section 3.6). The paper finishes with the comparison of the proposed algorithms with the state of the art (see Section 4) and the associated conclusions (Section 5).

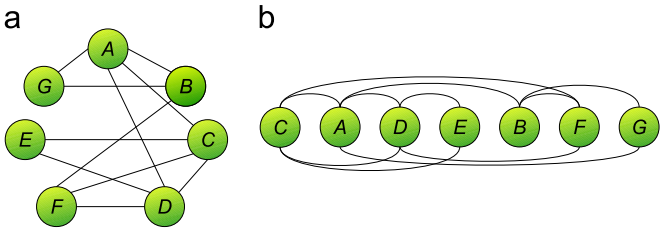

Fig. 1. (a) Graph illustrative example and (b) layout $\varphi = \{C, A, D, E, B, F, G\}$.

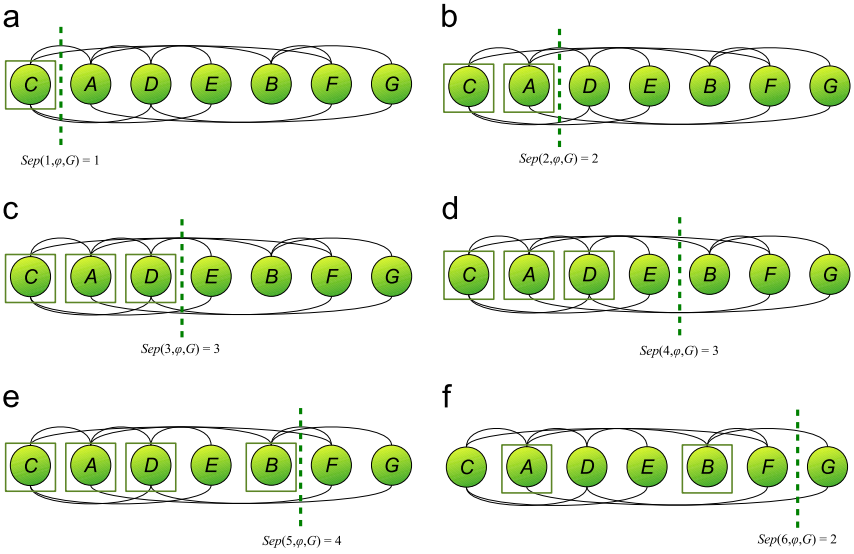

Fig. 2. (a)–(f) the computation of $Sep(p, \varphi, G)$ $(p = 1, \ldots, 6)$ for the layout $\varphi = \{C, A, D, E, B, F, G\}$.

2. Variable Neighborhood Search

The Variable Neighborhood Search (VNS) is a metaheuristic proposed by Mladenovic and Hansen [22] as a general framework to solve hard optimization problems. It is based on the idea of performing systematic changes of neighborhood structures within the search procedure. The original metaheuristic has been widely evolved with many extensions, highlighting Variable Neighborhood Descent (VND), Reduced VNS (RVNS), Basic VNS (BVNS), Skewed VNS (SVNS), General VNS (GVNS), Variable Neighborhood Decomposition Search (VNDS) and Reactive VNS. See [14,15] for recent thorough reviews.

Let N_k with $1 \leq k \leq k_{max}$ be a finite set of pre-selected neighborhood structures, where $N_k(x)$ is the set of neighbor solutions of x in the k-th neighborhood. When solving an optimization problem by using different neighborhood structures (N_k), VNS methodology proposes to explore them in three different ways: (i) random, (ii) deterministic, or (iii) mixed, which hybridizes both, deterministic and random.

2.1. Random exploration of neighborhoods

The Reduced Variable Neighborhood Search (RVNS) consists of exploring (generating) solutions at random in each N_k neighborhood. This variant does not consider the application of a local search procedure. In fact, the values of these random solutions are directly compared with the value of the incumbent solution, updating the best solution in case of improvement. The stopping criterion in this variant is usually the CPU time allowed (t_{max}).

The pseudo-code of RVNS is shown in Algorithm 1. It has three input arguments: an initial solution (x), the largest predefined neighborhood (k_{max}) and the maximum computing time (t_{max}). The procedure starts by performing a perturbation to the current solution using the function `Shake`, in step 5, and obtaining a new solution x'. In step 6, it is decided whether the RVNS needs to explore a larger neighborhood by increasing k (if x' is worse than x) or not, which implies to set $k=1$ (if x' is better than x). Steps 5 and 6 are repeated until k_{max} is reached. This parameter determines the maximum number of different neighborhoods to be explored in the current iteration when there is no improvement in the solution. Steps 3–8 are repeated until t_{max} is reached, starting in each iteration from the incumbent solution.

Algorithm 1. Pseudocode of RVNS.

```
1:  procedure RVNS (x, k_max, t_max)
2:  repeat
3:    k ← 1
4:  repeat
5:      x' ← Shake(x, k)
6:      (x, k) ← NeighborhoodChange(x, x', k)
7:    until k = k_max
8:    t ← CPUTime()
9:  until t > t_max
10: end RVNS
```

2.2. Deterministic exploration of neighborhoods

In Variable Neighborhood Descent (VND), several different neighborhoods are explored in order, typically from the smallest and fastest to evaluate, to the largest and slowest one. The process iterates over each neighborhood while improvements are found, performing local search until a local optimum is found at each neighborhood. Only strictly better solutions are accepted after each neighborhood search. Notice that most local search procedures are based on only one neighborhood which means that the local optimum is obtained with respect to that neighborhood. In VND, the returned solution is a local optimum in each N_k with $1 \leq k \leq k_{max}$. Therefore, the global optimum is more likely to be found.

The pseudo-code of VND is shown in Algorithm 2 where a nested strategy is considered. It has only two input arguments: an initial solution (x) and the number of neighborhoods (k_{max}). The procedure starts by obtaining a local optimum x' with respect to the first neighborhood. Then, instead of abandoning the search (as a local search procedure), VND resorts to the following neighborhood searching for an improvement. If so, the search starts again by considering the first neighborhood (which implies to set $k=1$). Otherwise, VND explores the next neighborhood by increasing k (until k_{max} is reached). This search strategy is condensed in steps 4 and 5 of Algorithm 2.

Algorithm 2. Pseudocode of VND.

```
1:  procedure VND(x, k_max)
2:  k ← 1
3:  repeat
4:    x' ← arg min_{y ∈ N_k(x)} f(y) {* Find the best neighbor in N_k(x)*}
5:    NeighborhoodChange(x, x', k)
6:  until k = k_max
7:  end VND
```

2.3. Mixed exploration of neighborhoods

In general terms, the RVNS strategy favors the diversification of the search, while the VND focuses on the intensification. Basic VNS (BVNS) and its generalization, known as General VNS (GVNS) are a compromise between these two strategies. The BVNS and the GVNS methods combine deterministic and random changes of neighborhoods, where the deterministic component is given by an improvement procedure and the random component is given by the shake procedure. The main difference between BVNS and GVNS is that the improvement strategy in BVNS is typically a local search, while in GVNS it is replaced by a VND algorithm. This approach has led to some of the most successful applications reported in the literature [16]. Therefore, we consider GVNS instead of BVNS.

Algorithm 3 shows the pseudo-code of GVNS. It starts by considering an initial solution x, which is an input argument together with k_{max} and t_{max}. This solution is then perturbed using the function `Shake` (step 5), obtaining a new solution x'. Then, a local optimum, x'', is reached by using the VND procedure (step 6). In step 7, it is decided whether the GVNS needs to explore a larger neighborhood (since x'' is worse than x) or not, which implies to set $k=1$. Steps 5–7 are repeated until k_{max} is reached. This parameter determines the maximum number of different neighborhoods to be explored in the current iteration. Steps 3–9 are repeated until t_{max} is reached, starting in each iteration from the incumbent solution.

Algorithm 3. Pseudocode of GVNS.

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

3. Algorithm approach

From an algorithmic perspective, VNS variants mainly differ in how they implement three basic strategies: shake (Section 3.3), neighborhood change (Section 3.4), and improvement (Section 3.6). We also consider a constructive procedure to create the initial solution needed for all VNS variants (Section 3.1). Finally, we propose different neighborhood structures (Section 3.2) and an efficient strategy to traverse them (Section 3.5).

3.1. Constructive procedure

We propose a new greedy procedure for constructing the initial solution for each VNS procedure in the context of the Vertex Separation problem. Given a graph G, this procedure is based on the creation of a tree T, where the nodes are organized in levels [21]. In this tree, two nodes belong to the same level if both have the same depth; otherwise, they belong to different levels. Let us denote $T = \{L_1, L_2, \ldots, L_l\}$, where the first one, L_1, contains only one vertex (i.e., the root of the tree). L_2 contains only the adjacent vertices to the one in L_1. In general, a level L_s with $1 < s \leq l$ contains all the vertices adjacent to some vertex in L_{s-1} that are not present in any previous level.

The constructed tree guarantees that the vertices in alternative levels are not adjacent. Therefore, we use a breadth-first search approach to construct the corresponding tree. The number of levels l (depth of the tree) exclusively depends on the vertex in L_1 and the graph G. In the context of the Vertex Separation problem, the larger the depth, the better the tree [8].

Algorithm 4 shows the pseudo-code of this procedure. The algorithm starts by constructing the set $\mathcal{T}$ with n different spanning trees of G. In particular, each $T \in \mathcal{T}$ is constructed by using a breadth-first search procedure (*BFS*), starting the search from a different vertex $v \in V$. Then, the algorithm finds the deepest tree $T^\star$ in the set $\mathcal{T}$ (see step 2). In the next step, we identify the set of levels in $T^\star$ and initialize the solution to the empty set (steps 3 and 4). The constructive procedure scans the levels L_i in ascending order (steps 5–10) and, for each element u in set L_i (see steps 6–9), the procedure inserts it in the best position of the partial solution under construction (steps 7 and 8). The constructive method ends when all the vertices of the tree have been inserted in the solution.

Algorithm 4. Pseudocode of the constructive procedure.

```
1:  procedure Constructive (G)
2:  T* ← arg max_{T ∈ 𝒯} Depth(BFS(G, T))
3:  {L_1, L_2, ..., L_l} ← Levels(T*)
4:  φ ← ∅
5:  for L_i ∈ T* do
6:    for u ∈ L_i do
7:      j* ← arg min_{1 ≤ j ≤ size(φ)} VS(Insert(φ, u, j))
8:      φ ← Insert(φ, u, j*)
9:    end for
10: end for
11: return φ
12: end Constructive
```

The proposed constructive procedure has two different stages. The algorithm first constructs a spanning tree of the graph, which places the vertices belonging to a given level in consecutive positions of the layout. This strategy tries to place adjacent vertices as close as possible in the corresponding layout. In general, when a vertex has its adjacents in close positions, it is expected that the involved vertices present low *Sep*-values. However, this first stage does not provide any information about the relative ordering among the adjacent vertices. Therefore, we refine the solution under construction in the second stage. Specifically, the constructive procedure searches, for each vertex, the best position to be placed in the partial solution.

3.2. Neighborhood structures

Solutions to graph arrangement problems are typically represented as permutations, where the first vertex in the permutation receives the label 1, the second vertex receives the label 2, and so on. In this section, we define two neighborhood structures for

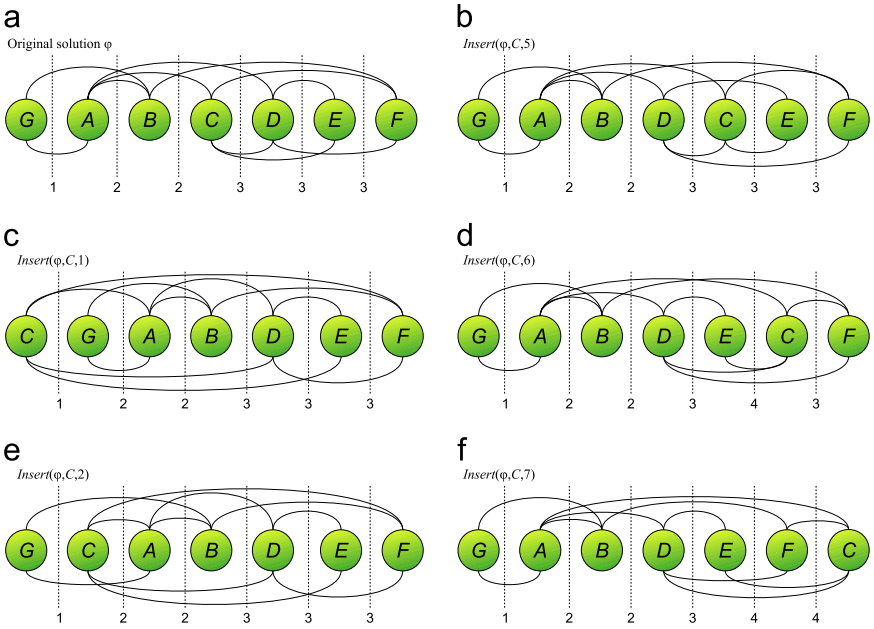

Fig. 3. (a) A given layout φ and (b)–(f) different solutions obtained inserting vertex C in positions $i = 1, 2, 5, 6$ and 7.

permutation-based solutions. They are based on insert moves, which are typically defined as follows: given a solution $\varphi = (v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_{j-1}, v_j, v_{j+1}, \ldots, v_n)$, $Insert(\varphi, v_i, j)$ consists of removing the vertex v_i from its current position $i = \varphi(v_i)$, and inserting it in position j, producing a new solution $\varphi' = (v_1 \ldots, v_{i-1}, v_{i+1}, \ldots, v_{j-1}, v_j, v_i, v_{j+1}, \ldots, v_n)$. For the sake of simplicity we denote $\varphi' = Insert(\varphi, v_i, j)$.

Considering the insert move defined above, we introduce the first neighborhood of φ as the set of solutions reachable by inserting the vertex v in each position of the permutation. This neighborhood is referred as $N_1(\varphi, v)$ and it is formally defined as

$$N_1(\varphi, v) = \{\varphi' = Insert(\varphi, v, j) : 1 \leq j \leq n\}.$$

The second neighborhood is based on the idea that we can identify "good" positions for a given vertex. Specifically, let $S(v)$ be the set of adjacent vertices to v. In mathematical terms, $S(v) = \{u \in V : (u, v) \in E\}$. Let $(u_1, u_2, \ldots, u_{|S(v)|})$ be the elements of $S(v)$ sorted in ascending order with respect to their label in the solution φ (i.e., $\varphi(u_1) < \varphi(u_2) < \cdots < \varphi(u_{|S(v)|})$). If we only consider moves of vertex v, we would obtain the maximum reduction of the *Sep*-value by inserting v in a position j such that $\varphi(u_1) < j < \varphi(u_2)$.

Let us consider that vertex v has a label $j = \varphi(v)$ such that $\varphi(u_1) < j < \varphi(u_2)$. We then expect that if we now insert v in a position $1 \leq j \leq \varphi(u_1)$ the value of the objective function can only remain equal or even get worse. Fig. 3 illustrates this behavior with the layout depicted in Fig. 3a. Consider the vertex C whose set of adjacent vertices is $S(\mathrm{C}) = \{\mathrm{A}, \mathrm{D}, \mathrm{E}, \mathrm{F}\}$, which satisfies the property described above (i.e., $\varphi(\mathrm{A}) < \varphi(\mathrm{C}) < \varphi(\mathrm{D}) < \varphi(\mathrm{E}) < \varphi(\mathrm{F})$) in $\varphi = \{\mathrm{G}, \mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}\}$. Fig. 3b and c shows the resulting solutions after the insertion of C in positions 1 and 2, respectively. The associated values of each solution produce a null improvement of the objective function.

If we alternatively insert the vertex v in a position j such that $\varphi(u_2) < j \leq n$, we observe again the same behavior (i.e., it obtains a null improvement or deteriorates the value of the objective function). Fig. 3d–f illustrates that the obtained solutions are equal or worse than the original φ in terms of quality.

Notice that this property is a heuristic rule, which means that positions $\varphi(u_1) < j < \varphi(u_2)$ are not always the best option to insert a vertex in terms of the value of $VS(\varphi, G)$. Fig. 4 illustrates an example where the described heuristic rule does not find the best position for the insertion of a vertex. Specifically, Fig. 4a shows an example of layout obtained from the graph depicted in Fig. 1a, with a *VS*-value of 4. If we consider the heuristic rule described above, and focusing on vertex E, the best position to insert it is in j such that $\varphi(\mathrm{C}) < j \leq \varphi(\mathrm{B})$. Then, $j = 3$ since $\varphi(\mathrm{C}) = 2$ and $\varphi(\mathrm{B}) = 3$. Fig. 4b shows the layout obtained after performing the aforementioned insert move, whose *VS*-value is 4. Therefore, there is no improvement in the move. However, considering again the layout depicted in Fig. 4a, if we insert the vertex E in the last position of the layout (position 7), we obtain the layout depicted in Fig. 4c with a *VS*-value of 3, which improves the original one and the layout obtained when applying the heuristic rule.

This neighborhood is mathematically defined as follows:

$$N_2(\varphi, v) = \{\varphi' = Insert(\varphi, v, j) : \varphi(u_1) < j < \varphi(u_2)\}.$$

In order to further reduce the size of this neighborhood we only consider to perform the insertion of v in only one position $\varphi(u_1) < j < \varphi(u_2)$ selected at random. Therefore, considering this reduction, the neighborhood $N_2(\varphi, v)$ finally contains only one solution.

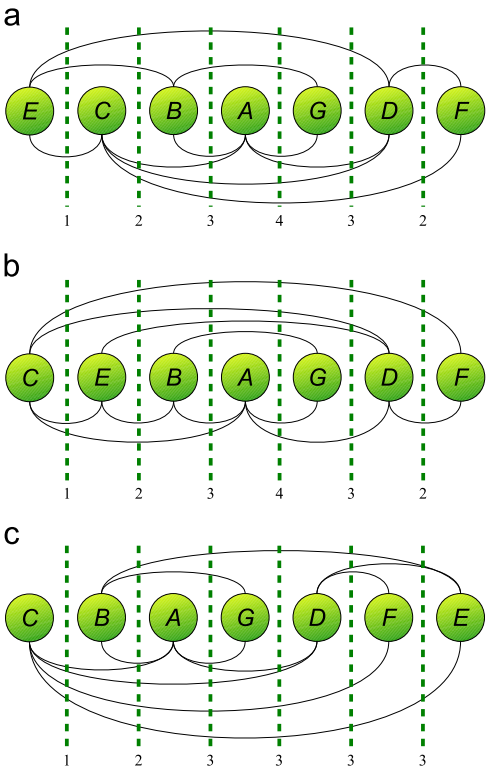

Fig. 4. (a) A given layout φ, (b) insertion of vertex E in the last position (7) starting from (a), and (c) insertion of vertex E in the most promising position (2) starting from (a).

3.3. Shake

In this section we propose 4 different shake functions for the vertex separation problem. These variants differ in how each one balances the intensification and the diversification. For each shake function we need to identify the set of vertices which will be perturbed and the associated positions for that vertices. In particular, $Shake{-}1(\varphi, k)$ selects k vertices at random and then those vertices are inserted again in positions selected at random. This method clearly favors the diversification of the search.

$Shake{-}2(\varphi, k)$ focuses on the intensification of the search. Specifically, this method selects the k vertices in V with the largest *Sep*-value in φ. Then, the procedure finds the best position to insert each vertex.

$Shake{-}3(\varphi, k)$ and $Shake{-}4(\varphi, k)$ try to find a compromise between the intensification and diversification. In particular, $Shake{-}3(\varphi, k)$ selects k vertices at random, but instead of inserting them in random positions, as $Shake{-}1(\varphi, k)$, they are inserted in the best position. Finally, $Shake{-}4(\varphi, k)$ selects the k vertices with the largest *Sep*-value in φ, and then the procedure inserts the vertices in random positions.

3.4. Neighborhood change

VSP is a min–max problem [8] where the value of the objective function is usually reached in several positions of the solution φ. This kind of problems presents a "flat landscape", which turns out in a challenge for classical local search procedures. Typically, local search strategies do not perform well from a computational point of view, since most of the moves have associated a null value. Then, given a graph G, changing the label i of a particular vertex v_i in φ (i.e., obtaining a new solution φ') such that its *Sep*-value is decreased, does not necessarily imply that $VS(G, \varphi') < VS(G, \varphi)$.

However, it can be considered as an interesting move if the number of vertices with a relative large *Sep*-value is reduced, regardless whether the objective function improves or not.

Given a solution φ, let us define $C(\varphi,i)$ as the set of vertices whose *Sep*-value is equal to i. For instance, the C sets for the layout depicted in Fig. 3a are: $C(\varphi,1)=\{G\}$, $C(\varphi,2)=\{A,B\}$, and $C(\varphi,3)=\{C,D,E\}$.

Let c_{max} be the index associated to the set which contains the vertices with the largest *Sep*-value. Obviously, $VS(\varphi,G)=c_{max}$. We propose an alternative formulation (i.e., a new objective function) for the vertex separation problem. This extended formulation, denoted as $EVS(\varphi,G)$ overcomes the lack of information provided by the original objective function. Specifically, this new objective function is defined as follows:

$$EVS(\varphi,G)=\sum_{i=1}^{c_{max}} n^i|C(\varphi,i)|.$$

With this new formulation, we could compare any pair of solutions beyond the value of c_{max}. In particular, we consider that a move improves the current solution (using the new objective function) if any vertex involved in the move is removed from $C(\varphi,i)$ and included in $C(\varphi,j)$ with $j<i$, and without increasing the cardinality of any set $C(\varphi,l)$ for $l>i$.

It is important to remark that if we have two solutions φ and φ' such that $VS(\varphi,G)<VS(\varphi',G)$ (i.e., φ is better than φ') then $EVS(\varphi,G)<EVS(\varphi',G)$. A formal proof of this property is provided as follows.

Proof. Let $c_{max}=VS(\varphi,G)$ and $c'_{max}=VS(\varphi',G)$ be the indexes associated to the set which contains the vertices with the largest *Sep*-value in φ and φ', respectively. By definition, the extended version of the objective function for a solution φ and a graph G is

$$EVS(\varphi,G)=\sum_{i=1}^{c_{max}} n^i|C(\varphi,i)|.$$

Let us separate the last term of the summation from the rest of elements

$$EVS(\varphi,G)=n^{c_{max}}|C(\varphi,c_{max})|+\sum_{i=1}^{c_{max}-1} n^i|C(\varphi,i)|.$$

By construction, the n vertices of the graph are distributed among the $C(\varphi,i)$ sets, where $1\le i\le c_{max}$. Obviously, if all vertices are in set $C(\varphi,c_{max})$, *EVS* presents its maximum value. In this situation $C(\varphi,c_{max})=n$ and $C(\varphi,i)=0$ for $1\le i<c_{max}$. Then,

$$EVS(\varphi,G)\le n^{c_{max}}\times n=n^{c_{max}+1}.$$

Our hypotheses establish that $c_{max}<c'_{max}$. Then, considering that both c_{max} and c'_{max} are integer values, it trivially holds that $c_{max}+1\le c'_{max}$. Therefore

$$EVS(\varphi,G)\le n^{c'_{max}}.$$

The extended version of the objective function for a solution φ' and a graph G is

$$EVS(\varphi',G)=\sum_{i=1}^{c'_{max}} n^i|C(\varphi',i)|.$$

If we again separate the last term of the summation from the rest of elements, we have

$$EVS(\varphi',G)=n^{c'_{max}}|C(\varphi',c'_{max})|+\sum_{i=1}^{c'_{max}-1} n^i|C(\varphi',i)|.$$

We can affirm that $|C(\varphi',c'_{max})|\ge 1$ since at least one vertex must be in $C(\varphi',c'_{max})$. Additionally, $\sum_{i=1}^{c'_{max}-1} n^i|C(\varphi',i)|\ge 0$ since the summation is performed over non-negative terms. Considering that the n vertices of the graph are distributed among the $C(\varphi',i)$ sets, where $1\le i\le c_{max}$, we analyze all the possible situations:

- There is only one vertex in $C(\varphi',c'_{max})$. Then, the remaining $n-1$ vertices are distributed among the $C(\varphi',i)$ sets where $1\le i<c'_{max}$. Therefore,

  $$EVS(\varphi',G)=n^{c'_{max}}+\sum_{i=1}^{c'_{max}-1} n^i|C(\varphi',i)|>n^{c'_{max}}\ge EVS(\varphi,G),$$

  which implies that $EVS(\varphi',G)>EVS(\varphi,G)$
- All the vertices are in $C(\varphi',c'_{max})$. Then, $|C(\varphi',i)|=0$, where $1\le i<c'_{max}$. Therefore,

  $$EVS(\varphi',G)=n^{c'_{max}}\times n>n^{c'_{max}}\ge EVS(\varphi,G),$$

  which implies that $EVS(\varphi',G)>EVS(\varphi,G)$
- $1<k<n$ vertices are in $C(\varphi',c'_{max})$. Then,

  $$EVS(\varphi',G)=n^{c'_{max}}\times k+\sum_{i=1}^{c'_{max}-1} n^i|C(\varphi',i)|>n^{c'_{max}}\ge EVS(\varphi,G),$$

  which implies that $EVS(\varphi',G)>EVS(\varphi,G)$

Then, independent of the distribution of the n vertices in the C-sets, we have proved that if $VS(\varphi,G)<VS(\varphi',G)$ then $EVS(\varphi,G)<EVS(\varphi',G)$.

We then propose a neighborhood change procedure that considers the aforementioned extended objective function. In particular, Algorithm 5 compares the incumbent solution φ with other solution φ' obtained from the k-th neighborhood using the *EVS*. If φ' is better than φ, the procedure updates the incumbent solution (step 3) and the search method resorts to the first neighborhood structure (step 4). Otherwise, the method considers the next neighborhood by increasing the value of k (step 6).

Algorithm 5. Pseudocode of the neighborhood change function.

```
1:  procedure NeighborhoodChange (φ, φ', k)
2:  if EVS(φ', G) < EVS(φ, G) then
3:    φ ← φ'
4:    k ← 1
5:  else
6:    k ← k+1
7:  end if
8:  end NeighborhoodChange
```

3.5. Efficient implementation of insert moves

Given a graph G and a layout φ, the computation of $VS(\varphi,G)$ requires to scan all the edges in the graph. Therefore, in a direct and straightforward implementation, the update of the objective function after performing a move is extremely time-consuming. However, it is clear that the *Sep*-values of some vertices does not change when we perform a move and therefore we do not need to re-compute it again. This idea was originally proposed in [8]. In particular, the authors proposed a move based on the interchange of two vertices in the layout. Considering the way in which the *Sep*-value is computed (difference between the label of incumbent vertex and the label of its adjacent with the largest label), if the move interchanges two vertices with labels i and j, it is only required to update the *Sep*-values from 1 to $\max\{i,j\}$.

We use in this paper an alternative definition of the *Sep*-value (see Section 1), which allows us to reduce the number of vertices that must be update. Specifically, after performing the move $Insert(\varphi,v_i,j)$, it is only required to update the *Sep*-values in

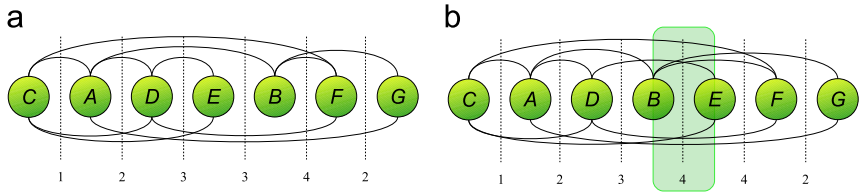

Fig. 5. Update of the objective function after a swap move.

positions k such that $i \leq k < j$. As it will be shown in Section 4, this observation saves a considerable amount of computing time. We propose an additional improvement in the way that the *Insert* move is implemented. In particular, the move is decomposed as a sequence of swaps, where each swap is defined as the interchange of vertices in consecutive positions in the corresponding permutation. This move is referred to us $\varphi' = Swap(\varphi, v_i, v_{i+1})$ in such a way that $\varphi'(v_i) = \varphi(v_{i+1})$ and $\varphi'(v_{i+1}) = \varphi(v_i)$.

Let us consider, without loss of generality, that $(i < j)$. Then, the move $Insert(\varphi, v_i, j)$ can be computed as the sequence of swaps of vertices in positions $(i, i+1)$, $(i+1, i+2)$ and so forth until $(j-1, j)$. If we store these values, the computation of the move $Insert(\varphi, v_i, j+1)$ is equivalent to compute the swap move between positions $(j, j+1)$ after performing $Insert(\varphi, v_i, j)$. The situation when $i > j$ is completely equivalent.

The computation of the objective function after a swap move between the vertex v_i in position $\varphi(v_i) = i$ and the vertex v_{i+1} in position $\varphi(v_{i+1}) = i+1$ only involves the update of the *Sep*-value in position i. Fig. 5 shows an example of a swap move between vertices B and E, in positions 4 and 5, respectively. It is easy to see that the only *Sep*-value that needs to be updated is the one in position 4 (highlighted in Fig. 5b). In particular, $Sep(4, \varphi, G) = 3$ and, after performing the swap, $Sep(4, \varphi', G) = 4$, while the other *Sep*-values remain unaltered.

In addition, the value of $Sep(i, \varphi', G)$ can be incrementally computed by considering the value of $Sep(i-1, \varphi, G)$. In particular, $Sep(i, \varphi', G) = Sep(i-1, \varphi, G) + \delta^+ - \delta^-$, where δ^+ determines if the vertex in position i (previously in position $i+1$) has, at least, one adjacent in the set $R(i, \varphi, G)$. If so, the *Sep*-value is incremented by one unit. Otherwise, it holds the same value. On the other hand, δ^- refers to those vertices in $L(i, \varphi, G)$ whose adjacent vertex with the largest label is the one in position i. In mathematical terms

$$\delta^+ = \begin{cases} 1 & \text{if } |R(i, \varphi, G) \cap S(v)| > 0, \\ 0 & \text{otherwise} \end{cases}$$

$$\delta^- = \left| \left\{ v \in L(i, \varphi, G) \cap S(v) : \max_{u \in S(v)} \varphi(u) = i \right\} \right|,$$

where $S(v)$ represents the set of adjacent vertices to v. This strategy saves a considerable CPU time since it only updates one position and the computation of the *Sep*-value in that position is incrementally performed.

Considering again the example shown in Fig. 5, $\delta^+ = 1$ because the vertex B in position 4 has one or more adjacent vertices (F and G) placed in a position larger than 4. On the other hand, $\delta^- = 0$ since there are not vertices in the solution whose adjacent vertex with the largest label is B.

3.6. Local search methods

In this paper we propose two local search strategies, LS_1 and LS_2, that scan the neighborhoods N_1 and N_2, respectively. Algorithm 6 represents the pseudocode for both methods, where $LS_1 = LocalSearch(\varphi, 1)$ and $LS_2 = LocalSearch(\varphi, 2)$. The pseudocode starts by sorting the vertices in descending order of their *Sep*-value, constructing the ordered set A (step 7). Then, *LocalSearch* finds the best solution for each node of A in their corresponding neighborhood (step 9). Next, the new solution φ' is compared with the best solution so far $\varphi^\star$ considering the alternative objective function (i.e., *EVS*) in step 10, updating it if needed (step 11). The *LocalSearch* procedure performs moves while an improvement is produced (steps 4–15). Finally, the method returns the best solution found in the corresponding neighborhood.

Algorithm 6. Pseudocode of the local search procedure.

```
1:  procedure LocalSearch(φ, k)
2:  improvement ← true
3:  φ* ← φ
4:  while improvement do
5:    improvement ← false
6:    φ ← φ*
7:    A ← OrderBySepValue(φ)
8:    for all v ∈ A do
9:      φ' ← arg min_{φ ∈ N_k(v)} EVS(φ, G)
10:     if EVS(φ', G) < EVS(φ*, G) then
11:       φ* ← φ'
12:       improvement ← true
13:     end if
14:   end for
15: end while
16: return φ*
17: end LocalSearch
```

4. Computational experience

This section reports the computational experiments that we have performed for testing the efficiency of the proposed three VNS variants (RVNS, VND, and GVNS) for solving the VSP. All the algorithms were implemented in Java SE 6 and the experiments were conducted on an Intel Core i7 2600 CPU (3.4 GHz) and 4 GB RAM. We have considered three sets of instances previously used in this problem. All instances are available at http://www.optsicom.es/vsp/. A detailed description of each type of instances follows:

- HB: We derived 62 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 $M_{uv} \neq 0$. From the original set we have selected the 62 graphs with $n \leq 1000$. The number of vertices and edges ranges from 24 to 960 and from 34 to 3721, respectively.
- Grids: This set consists of 50 matrices constructed as the Cartesian product of two paths [26]. They are also called two-dimensional meshes and the optimal solution of the VSP for squared grids is known by construction, see [6]. Specifically, the vertex separation value of a square grid of size $\lambda \times \lambda$ is λ. For this set, the vertices are arranged on a square grid with a

dimension $\lambda \times \lambda$ for $5 \leq \lambda \leq 54$. The number of vertices and edges range from $5 \times 5 = 25$ to $54 \times 54 = 2916$ and from 40 to 5724, respectively.

- `Trees`: Let $T(\lambda)$ be set of trees with minimum number of nodes and vertex separation equal to λ. As it is stated in [10], there is just one tree in $T(1)$, namely the tree with a single edge, and another one in $T(2)$, the tree constructed with a new node acting as root of three subtrees that belong to $T(1)$. In general, to construct a tree with vertex separation $\lambda+1$ it is necessary to select any three members from $T(\lambda)$ and link any one node from each of these to a new node acting as the root of the new tree. The number of nodes, $n(\lambda)$, of a tree in $T(\lambda)$ can be obtained using the recurrence relation $n(\lambda) = 3n(\lambda-1)+1$ where and $n(1) = 2$ (see [10] for additional details). We consider 50 different trees: 15 trees in $T(3)$, 15 trees in $T(4)$ and 20 trees in $T(5)$. The number of vertices and edges range from 22 to 202 and from 21 to 201, respectively.

We have divided our experimentation into two different parts: preliminary experimentation and final experimentation. The preliminary experiments were performed to set the values of the key search parameters of each VNS variant as well as to show the merit of the proposed search strategies. We consider a representative subset of 23 `HB` instances, with different sizes and densities. Specifically, we consider: `494_BUS`, `662_BUS`, `685_BUS`, `BCSPWR04`, `CAN_292`, `CAN_445`, `CAN_634`, `CAN_715`, `DWT_245`, `DWT_307`, `DWT_310`, `DWT_361`, `DWT_419`, `DWT_503`, `DWT_592`, `DWT_758`, `LHSP_778`, `NOS3`, `NOS4`, `NOS5`, `NOS6`, `NOS7`, and `PLAT362`.

In our first experiment we compare the performance of the proposed constructive procedure (named as C_{greedy} and described in Section 3.1) with the best previous constructive approach (named as C2 in [8]). We generate one solution for each instance with each constructive procedure. Table 1 reports #Best, number of best solutions found in the experiment; Avg., average quality over all instances; Dev. (%), average percent deviation with respect to the best solution found in the experiment; and Time, average computing time in seconds required by the procedure. We report these statistics in the remaining experiments.

Table 1 shows that C_{greedy} clearly outperforms C2 in terms of both, number of best (23 versus 8) and average percentage deviation (0.00% versus 11.88%). Our constructive procedure needs more CPU time (0.53 versus 0.18). However, this time is negligible when considering the computing time of the whole algorithm (constructive procedure plus the corresponding VNS variant). Therefore, we consider C_{greedy} as the best constructive procedure and it will be used for the remaining experiments.

In the next experiment, we study the performance of the incremental computation of the objective function described in Section 3.5 over the instances used in the preliminary experimentation. In particular, we test whether the use of swap moves (as a way of implementing insert moves) reduces the CPU time or not. We construct a solution with C_{greedy} and then we improve it with the local search based on insert moves. In order to have a clear idea about the saving in the computing time, we compare our proposal with the one proposed in [8], and with a direct computation of the objective function. Fig. 6 depicts a diagram where the *X*-axis represents the set of instances considered for this experiment (ordered according to the number of vertices) and the *Y*-axis gives the computing time required to obtain a local optimum for the three considered methods (Insert Moves, Interchange Moves, and Direct) in the corresponding instance. The *Y*-axis uses a logarithmic scale to reduce the ranges to a more manageable size. The figure clearly shows that the saving in computing time is significant for the introduced method. Specifically, for these instances the proposed method is about 40 times faster than the incremental computation described in [8] and it is about 1000 times faster than the direct computation of the objective function. Notice that the

Table 1
Comparison of different constructive procedures.

	C_{greedy}	C2
#Best	23	8
Avg.	42.26	46.43
Dev. (%)	0.00	11.88
Time	0.53	0.18

Table 2
Comparison of different shake procedures.

	RVNS-1	RVNS-2	RVNS-3	RVNS-4
#Best	15	7	19	7
Avg.	37.74	42.09	37.48	42.04
Dev. (%)	2.33	16.24	1.27	16.06
Time	101.96	102.00	101.99	101.96

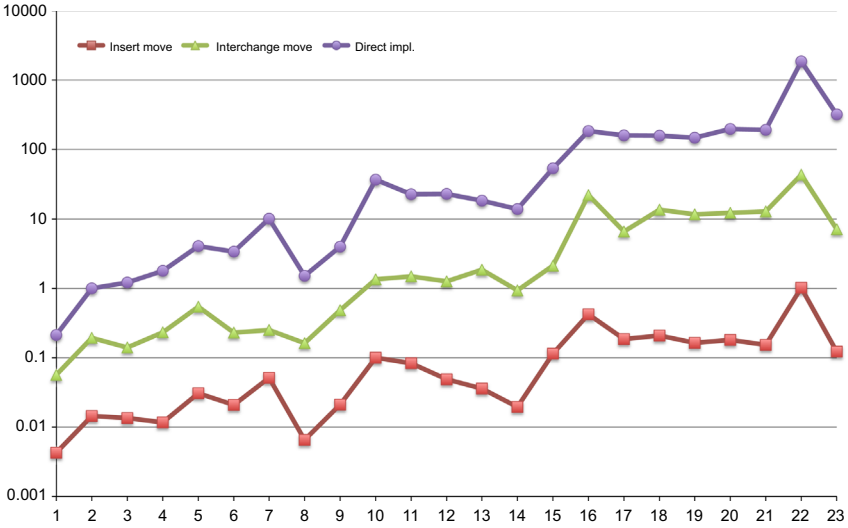

Fig. 6. Incremental computation of the objective function.

Table 3
Comparison of different neighborhoods.

	LS_1	LS_2	VND
#Best	17	6	23
Avg.	37.35	41.13	36.65
Dev. (%)	2.59	15.01	0.00
Time	3.93	0.78	6.94

incremental computation described in [8] is about 25 times faster than the direct computation.

In the next experiment, we compare the performance of 4 RVNS variants (RVNS-1, RVNS-2, RVNS-3, and RVNS-4). The only difference among them is the shake procedure (see Section 3.3) used in each one. In particular, RVNS-x considers the $Shake{-}x$ function, where $x \in \{1, 2, 3, 4\}$. As it is pointed out in [15], the best outcomes in RVNS are obtained when the value of k_{max} is 2 or 3. Then, we select $k_{max} = 3$ since we obtained slightly better results. Table 2 shows the results of the 4 RVNS variants when t_{max} is set to 100 s. In this time horizon, RVNS-3 emerges as the best procedure, where the shake procedure selects vertices at random that are then placed in their best position (maximum decreasing of the objective function). This strategy seems to be the most promising one since it balances diversification (random selection of vertices) with intensification (greedy selection of the best position). In fact, selecting vertices at random (RVNS-1 and RVNS-3) produces better outcomes than selecting them in a greedy fashion (RVNS-2 and RVNS-4).

In the fourth preliminary experiment, we compare the performance of the VND with the local search procedures in isolation (i.e., LS_1 and LS_2). Typically, VND explores the neighborhoods from the smallest and fastest to evaluate, to the slowest and largest one. Consequently, our VND first considers N_2 and then N_1. Notice that the neighborhoods are explored in a nested strategy. Results in Table 3 confirm that the VND procedure compares favorably with simple local search methods. Specifically, VND achieves the lowest deviation (0.00%) compared with the two local search methods tested (2.59% and 15.01% for LS1 and LS2, respectively). It is worth to mention that the results of LS2 seem to be quite bad. However, when coupling LS1 and LS2 in a VND strategy, the resulting algorithm obtains the best outcomes.

The next experiment consists of selecting the best GVNS variant by running a single full-factorial experiment. In particular, we consider the 4 shake functions described in Section 3.3 and $k_{max} = \{0.05n, 0.10n, 0.15n, 0.20n\}$, n being the number of vertices of the instance. For the sake of simplicity we do not show this experiment in a table and we only report that the best results are obtained using the $Shake{-}1$ procedure. This result is in line with the ones shown in Table 2. Specifically, the best strategy consists of selecting the vertices at random. However, in the GVNS context, the positions (where those vertices are placed) are selected at random. This result can be partially explained when considering that the shake procedure in GVNS mainly diversify the search and the intensification is performed by the improvement strategy (i.e., VND algorithm). As expected, the higher the value of k_{max}, the better the results are. Unfortunately, the computing time also increases with the value of k, so we set $k = 0.15n$ as a compromise between computing time and quality.

We compare in the next experiment the best of each VNS variant (RVNS, VND, and GVNS) with the best previous method identified in the state of the art [8], which is also a VNS algorithm. Consequently, in order to avoid a misunderstanding with the VNS variants that we are proposing in this paper, we refer to this method as BestPrev. To provide a fair comparison, GVNS has the same stopping criterion than BestPrev (either k reaches k_{max} or the CPU time exceeds $t_{max} = 1000$ s). RVNS has t_{max} as input parameter, which is set to the same CPU time than BestPrev. Finally, in the case of VND, t_{max} is not an input parameter. Therefore, to be able to execute VND for the same CPU time than the other variants, we consider a multi-start scheme, where each iteration consists of a construction (with the procedure described in Section 4) and then a VND execution. Table 4 reports the results of this experiment. This table is structured into three main rows. Each row represents the results associated to each set of instances. In particular, the first row reports the results of the four algorithms over the set `Grids`. In sight of these results we can conclude that grids instances are easily solved by the four procedures. Although the computing time seems to be relatively large, it is important to note that all the procedures find the optimum value in less than a second. Therefore, we do believe that this set of instances does not worth to be included in future comparisons. Regarding the instances of the set `Trees` (second main row), the four algorithms are able to find the optimal solution for the small and medium instances (i.e., Trees with 22 and 67 vertices). Considering the largest tree instances (with 202 vertices), the BestPrev only obtains one optimum solution, while the three proposed methods obtain, 4 (RVNS), 5 (MS_VND) and 10 (GVNS) out of 20, respectively. Although the CPU time of the four procedures is relatively small, our three methods need much less time to produce better results.

Table 4
Final comparison over the sets `Grids` (50 instances), `Trees` (50 instances), and `HB` (62 instances).

		GVNS	RVNS	MS_VND	BestPrev
`Grids`	#Opt.	50	50	50	50
	Avg.	29.5	29.5	29.5	29.5
	Dev. (%)	0.00	0.00	0.00	0.00
	Time	93.32	215.30	101.08	1422.76
`Trees`	#Opt	40	34	35	31
	Avg.	4.30	4.46	4.40	4.64
	Dev. (%)	4.00	7.00	6.00	11.00
	Time	9.07	21.10	2.98	126.73
`HB`	#Best	53	28	27	34
	Avg.	24.60	27.00	26.77	26.11
	Dev. (%)	2.07	10.85	11.91	6.90
	Time	482.08	700.03	702.89	705.60

The third main row reports the results over the most challenging set of instances (i.e., `HB`). As it was aforementioned, the optimum of these instances is not known. Moreover, the properties of each instance (density, max and min degree, etc.) enormously vary from one to another. Table 4 shows that GVNS clearly outperforms the other three methods (including the best method found in the state of the art) in average percentage deviation and number of best solutions found. Moreover, it only needs a half of the computing time used by the other three methods to produce much better results. Therefore, results reported in this table suggest the superiority of GVNS over the other three methods. Regarding RVNS and MS_VND, BestPrev apparently obtains better results. In order to confirm these hypothesis, we applied the non-parametric Friedman test [13] for multiple correlated samples to the best solutions obtained by GVNS, RVNS, MS_VND, and BestPrev. This test computes, for each instance, the rank value of each method according to 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 p-value or significance will be small. We only consider in this experiment the instances in the `HB` set, and the largest instances of the `Tree` set (trees with 202 vertices), since grids and small and medium trees are optimally solved by all the algorithms.

The resulting p-value of 0.00001 (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 1.88 (GVNS), 2.61 (BestPrev), 2.75 (MS_VND), and 2.76 (RVNS). To detect the differences among BestPrev and the three proposed methods in this paper and considering the proximity in the rank between BestPrev, RVNS and MS_VND, we conduct a Wilcoxon's test. Let $R+$ be the sum of ranks for the functions on which the first algorithm outperforms the second one, and $R-$ be the sum of ranks for the opposite. Ranks corresponding to zero differences are split evenly among the sums. If $\min\{R+, R-\}$ is less than or equal to the critical value, the Wilcoxon's test detects significant differences between the algorithms, which means that an algorithm outperforms its opponent.

Table 5 summarizes the results of this statistical test with a level of significance 0.05, where the values of $R+$ (associated to one of our methods) and $R-$ (associated to BestPrev) of the test are specified. The third column reports the p-value associate to each experiment and the last column indicates whether Wilcoxon's test found statistical differences between these algorithms or not. In particular, if $\min\{R+, R-\}$ is less than or equal to the critical value [29] in this experiment, this test detects that an algorithm outperforms its opponent. In particular, if this fact occurs and, simultaneously, $R- = \min\{R+, R-\}$, then our method is better than BestPrev. However, the confidence of the test is always determined by the p-value.

These results complement the ones reported in Table 4. Specifically, the Wilcoxon's test again confirm the superiority of GVNS over BestPrev. Additionally, this test shows that there is no significant statistical differences between RVNS versus BestPrev and MS_VND versus BestPrev. Therefore, even considering the results reported in Tables 4 and 5, it is not possible to say that BestPrev is statistically better than either RVNS or MS_VND.

In the final experiment we explore the behavior of these methods over a long-term time horizon, we run GVNS, RVNS, MS_VND and BestPrev for 30 min, reporting every 30 s the average deviation of the best solution found. We select the ten largest instances in HB to illustrate the performance of the four procedures over the hardest instances. Fig. 7 shows the corresponding average time profile, where it is represented with a dash line the best results found in the state of the art. We can observe that GVNS consistently produces the best results, from the very beginning to the end of the experiment. It is important to remark that GVNS produces high quality solutions in the short-term horizon (outperforming the state of the art in the first 30 s) and in the long-term horizon (producing considerable improvements about 1500 s). This fact shows that the intensification stage (VND procedure) produces good solutions in short CPU time, but also the diversification stage (shake procedure) is able to find promising regions in the search space.

MS_VND presents competitive results only in the short-time horizon. This fact can be partially explained by considering that the constructive procedure is not designed to perform multiple constructions. Consequently, it produces high quality but not diverse solutions. The improvement of the RVNS algorithm is constant during the 30 min. As it was pointed out above, the shake procedure embedded in this algorithm presents a balance between diversification (selecting vertices at random) and intensification (inserting them in the best possible position). This strategy seems to be successful over the whole experiment. In fact, it is expected that the RVNS would outperform MS_VND in larger computing times. It is worth to mention that the superiority of GVNS over the remaining methods comes from the fact that GVNS behaves like MS_VND in short CPU times, while it behaves like RVNS in larger computing times.

Table 5
Wilcoxon's test results.

	$R-$	$R+$	p-value	Significant
GVNS versus BestPrev	134.5	900.5	0.00	Yes
RVNS versus BestPrev	520.5	299.5	0.14	No
MS_VND versus BestPrev	684.5	491.5	0.32	No

Table 6
Friedman's test results over the ten largest instances.

	GVNS	RVNS	MS_VND	BestPrev	p-value
1 min	1.90	3.15	1.90	3.05	0.034
10 min	1.80	2.90	2.10	3.20	0.049
20 min	1.75	2.80	2.20	3.25	0.048
30 min	1.50	2.60	2.45	3.45	0.009

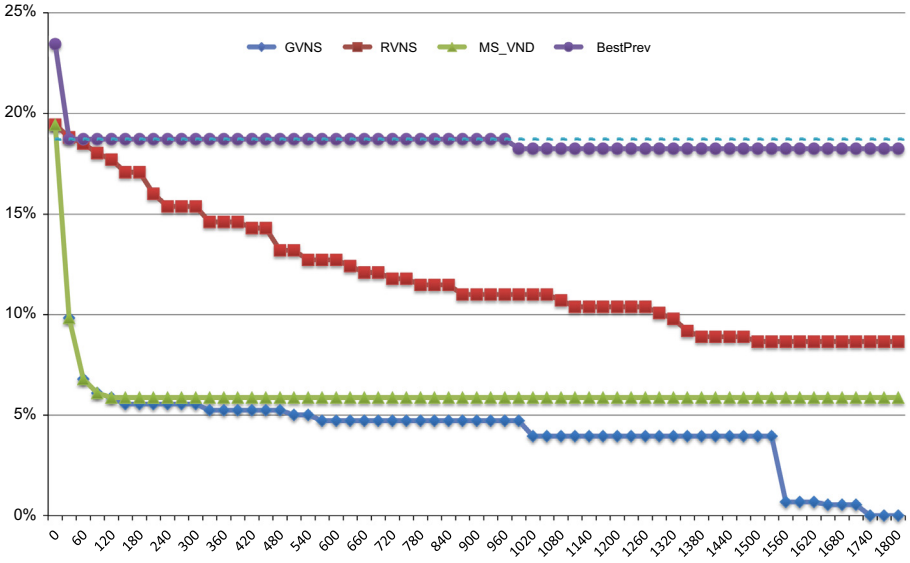

Fig. 7. Search profile for a 30 min run on the largest instances.

Analyzing the results obtained by BestPrev, we can determine that this procedure is not suitable when facing large instances. The algorithm starts improving at the very beginning (60 s). After that the method gets stuck until reaching 1000 s, mainly because the local search method is slower than the ones proposed in this paper. As a consequence, this method presents the worst performance (at least in these instances).

Finally, we analyze the results of this experiment by considering the Friedman's test. Specifically, we apply a Friedman's test in four different time stamps: 1 min, 10 min, 20 min, and 30 min. Table 6 shows the rankings when considering the 10 largest (and hardest) instances of the HB set. The resulting p -value in all the considered time stamps indicates that the differences are statistically significant. Specifically, for 1 min GVNS and MS_VND emerge as the best methods (with a ranking 1.90), followed by BestPrev (3.05) and RVNS (3.15). When considering the results after 10 and 20 min, the GVNS improves its ranking as well as RVNS. It is interesting to mention that MS_VND and BestPrev seem to be stuck so their ranking get worse. Finally, after 30 min, the GVNS even increases more the difference with respect to the other algorithms (1.50). MS_VND holds the second position according to the ranking value (2.45), but closely followed by the RVNS (third method with ranking 2.60). Finally, BestPrev systematically produces the worst results, which led it to the last position of the ranking with a value of 3.45.

5. Conclusions

In this paper, we propose different methods based on the Variable Neighborhood Search methodology to deal with the Vertex Separation problem. We provide an extensive experimental comparison among them and with best previous method in the state of the art. In more detail, we propose a constructive procedure, four shake methods (with different balance between intensification and diversification), two new neighborhood structures (and efficient strategies to explore them), and an extended version of the objective function for the considered problem which allows the comparison of solutions beyond the original objective function. We embed these strategies in a Reduced Variable Neighborhood Search (RVNS), a Variable Neighborhood Descent (VND) and a General Variable Neighborhood Search (GVNS). We performed an extensive computational testing over a set of 162 instances, considering different time horizons. Experimental results show that the proposed algorithms outperform the best method identified in the state of the art. We also proved statistical tests to confirm the significance of the obtained results, and GVNS emerges as the best algorithm in terms of quality in any set of instances and time horizon.

According to our experimentation, the Grids and Trees instances can be considered "easy to solve". In our opinion, these instances should no longer be considered for future studies since they do not allow us to evaluate the actual performance of the compared methods. On the contrary, the Harwell–Boeing instances are actually a real challenge for modern heuristic methods. Therefore, we recommend this set of instances to be used as the benchmark for further experimental studies.

Of particular interest in our work has been testing the combination of diversification and intensification strategies in the context of VNS. Through extensive experimentation, we have been able to determine the benefits of this combination. We purposefully added these mechanisms in order to measure their effects and studied the combinations that resulted in effective solution procedures with improved outcomes. We believe that our findings can be translated to other combinatorial problems and it will help in the development of more elaborated VNS methods.

Acknowledgments

This research has been partially supported by the Spanish Ministry of "Economía y Competitividad", Grants ref. TIN2011-28151, TIN2012-35632-C02, and the Government of the Community of Madrid, Grant ref. S2009/TIC-1542.

References

[1] Bodlaender HL, Möhring RH. The pathwidth and treewidth of cographs. In: Proceedings of the second Scandinavian workshop on algorithm theory. SWAT'90; 1990. p. 301–9.
[2] Bodlaender HL, Kloks T, Kratsch D. Treewidth and pathwidth of permutation graphs. SIAM J Discrete Math 1995;8(4):606–16.
[3] Bodlaender HL, Gilbert JR, Hafsteinsson H, Kloks T. Approximating treewidth, pathwidth, frontsize, and shortest elimination tree. J Algorithms 1995;18(2):238–55.
[4] Bodlaender HL, Gustedt J, Telle JA. Linear-time register allocation for a fixed number of registers. In: Proceedings of the symposium on discrete algorithms; 1998. p. 574–83.
[5] Bollobás B, Leader I. Edge-isoperimetric inequalities in the grid. Combinatorica 1991;11:299–314.
[6] Díaz J, Petit J, Serna M. A survey of graph layout problems. ACM Comput Surv 2002;34(3):313–56.
[7] Duarte A, Martí R, Resende MGC, Silva RMA. GRASP with path relinking heuristics for the antibandwidth problem. Networks 2011;58(3):171–89.
[8] Duarte A, Escudero LF, Mart R, Mladenovic N, Pantrigo JJ, Snchez-Oro J. Variable neighborhood search for the vertex separation problem. Comput Oper Res 2012;39:3247–55.
[9] Dujmović V, Fellows MR, Kitching M, Liotta G, Mccartin K, Nishimura N, et al. On the parameterized complexity of layered graph drawing. Algorithmica 2008;52(2):267–92.
[10] Ellis JA, Sudborough IH, Turner JS. The vertex separation and search number of a graph. J Inf Comput 1994;113:50–79.
[11] Ellis JA, Markov M. Computing the vertex separation of unicyclic graphs. Inf Comput 2004;192(2):123–61.
[12] Fellows MR, Langston MA. On search, decision and the efficiency of polynomial-time algorithms. J Comput Syst Sci 1994;49(3):769–79.
[13] Friedman M. A comparison of alternative tests of significance for the problem of m rankings. Ann Math Stat 1940;11:86–92.
[14] Hansen P, Mladenovic N, Moreno JA. Variable neighbourhood search: methods and applications. Ann Oper Res 2010;175(1):367–407.
[15] Hansen P, Mladenovic N, Brimberg J, Moreno-Pérez JA. Variable neighbourhood search. Handbook of Metaheuristics 2010;146:61–86.
[16] Aleksandar Ilić, Dragan Urošević, Jack Brimberg, Nenad Mladenovic. A general variable neighborhood search for solving the uncapacitated single allocation p-hub median problem. Eur J Oper Res 2010;206(2):289–300.
[17] Kinnersley NG. The vertex separation number of a graph equals its path-width. Inf Process Lett 1992;42(6):345–50.
[18] Kirousis M, Papadimitriou CH. Interval graphs and searching. Discrete Math 1985;55(2):181–4.
[19] Kirousis M, Papadimitriou CH. Searching and pebbling. Theory Comput Sci 1986;47(2):205–18.
[20] Leiserson CE. Area-efficient graph layouts (for VLSI). In: Proceedings of the IEEE symposium on foundations of computer science; 1980. p. 270–81.
[21] Lewis JG. The Gibbs–Poole–Stockmeyer and Gibbs–King algorithms for reordering sparse matrices. ACM Trans Math Software 1982;8:190–4.
[22] Mladenović N, Hansen P. Variable neighborhood search. Comput Oper Res 1997;24:1097–100.
[23] Pantrigo JJ, Martí R, Duarte A, Pardo EG. Scatter search for the cutwidth minimization problem. Ann Oper Res 2012;199:285–304.
[24] Peng SL, Ho C-W, Hsu TS, Ko MT, Tang CY. A linear-time algorithm for constructing an optimal node-search strategy of a tree. In: Proceedings of the 4th annual international conference on computing and combinatorics, COCOON '98; 1998. p. 279–88.
[25] Pi nana E, Plana I, Campos V, Martí R. GRASP and Path relinking for the matrix bandwidth minimization. J Oper Res 2004;153:200–10.
[26] Raspaud A, Schröder H, Sýkora O, Török L, Vrt'o I. Antibandwidth and cyclic antibandwidth of meshes and hypercubes. Discrete Math 2009;309:3541–52.
[27] Rodriguez-Tello E, Jin-Kao H, Torres-Jimenez J. An effective two-stage simulated annealing algorithm for the minimum linear arrangement problem. Comput Oper Res 2008;35(10):3331–46.
[28] Skodinis K. Computing optimal strategies for trees in linear time. In: Proceedings of the 8th annual European symposium on algorithms; 2000. p. 403–14.
[29] Wilcoxon F. Individual comparisons by ranking methods. Biometrics 1945;1:571–95.