Computers & Operations Research 39 (2012) 3247–3255

Contents lists available at SciVerse ScienceDirect

Computers & Operations Research

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

Variable neighborhood search for the Vertex Separation Problem

Abraham Duarte [a,*], Laureano F. Escudero [b], Rafael Martí [c], Nenad Mladenovic [d], Juan José Pantrigo [a], Jesús Sánchez-Oro [a]

[a] Dpto. de Ciencias de la Computación, Universidad Rey Juan Carlos, Móstoles, Madrid, Spain
[b] Dpto. de Estadística e Investigación Operativa, Universidad Rey Juan Carlos, Móstoles, Madrid, Spain
[c] Dpto. de Estadística e Investigación Operativa, Universidad de Valencia, Valencia, Spain
[d] Department of Mathematics, SISCM, London, United Kingdom

ARTICLE INFO

Available online 19 April 2012

Keywords:
Combinatorial optimization
Metaheuristics
Variable neigborhood search
Layout problems

ABSTRACT

The Vertex Separation Problem 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. This optimization problem is strongly related to other well-known graph problems; such as the Path-Width, the Node Search Number or the Interval Thickness, among others. All of these optimization problems are NP-hard and have practical applications in VLSI (Very Large Scale Integration), computer language compiler design or graph drawing. Up to know, they have been generally tackled with exact approaches, presenting polynomial-time algorithms to obtain the optimal solution for specific types of graphs. However, in spite of their practical applications, these problems have been ignored from a heuristic perspective, as far as we know. In this paper we propose a pure 0-1 optimization model and a metaheuristic algorithm based on the variable neighborhood search methodology for the Vertex Separation Problem on general graphs. Computational results show that small instances can be optimally solved with this optimization model and the proposed metaheuristic is able to find high-quality solutions with a moderate computing time for large-scale instances.

© 2012 Elsevier Ltd. All rights reserved.

1. Introduction

Let $G(V,E)$ be an undirected graph where V $(n=|V|)$ and E $(m=|E|)$ are the sets of vertices and edges, respectively. A linear layout φ of the vertices of G is a bijection or mapping $\varphi : V \rightarrow \{1,2,\ldots,n\}$ in which each vertex receives a unique and different integer between 1 and n. For vertex u, let $\varphi(u)$ denote its position or label in layout φ. 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,

$$L(p,\varphi,G)=\{v \in V : \varphi(v) \le p\} \quad \text{and} \quad R(p,\varphi,G)=\{v \in V : \varphi(v) > p\}.$$

Since layouts are usually represented in a straight line, where the vertex in position 1 comes first, $L(p,\varphi,G)$ can be simply called the set of left vertices with respect to position p and, $R(p,\varphi,G)$ the set of right vertices w.r.t. p.

The *Cut*-value at position p of layout φ, $Cut(p,\varphi,G)$, is defined as the number of vertices in $L(p,\varphi,G)$ with one or more adjacent vertices in $R(p,\varphi,G)$, then,

$$Cut(p,\varphi,G)=\left|\{u \in L(p,\varphi,G) : \exists v \in R(p,\varphi,G) \cap N(u)\}\right|,$$

where $N(u)=\{v \in V : (u,v) \in E\}$. The Vertex Separation value (VS) of layout φ is the maximum of the *Cut*-value among all positions in layout φ : $\mathrm{VS}(\varphi,G)=\max_p Cut(p,\varphi,G)$.

The Vertex Separation Problem (VSP) consists of finding a layout, say φ^*, minimizing the VS in graph G. Despite of its practical applications, there is no previous heuristic or metaheuristic algorithm to find good solutions in short computing times. We propose a variable neighborhood search (VNS) [18] approach whose performance is assessed by a broad testbed of instances.

The remainder of this paper is organized as follows. Section 2 presents the Vertex Separation Problem, including its application domain. Section 3 formalizes the mathematical optimization model. Section 4 presents our algorithmic approach based on the VNS metaheuristic framework. Section 5 reports on an extensive computational experience to validate the proposed algorithm by (1) comparing its performance and computing time versus the optimization of the mathematical model for small instances and (2) analyzing its performance for large instances. Finally, Section 6 summarizes the main conclusions of our research.

2. Description, related problems and applications

The Vertex Separation Problem (VSP) in graph G consists of finding a layout, say φ^*, that minimizes $\mathrm{VS}(G,\varphi)$, where $\mathrm{VS}(G,\varphi)$ is

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

0305-0548/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.cor.2012.04.017

the maximum *Cut*-value in graph G for layout φ, i.e., $\mathrm{VS}(G,\varphi)=\max_p Cut(p,\varphi,G)$. For the sake of simplicity, we denote the optimum value $\mathrm{VS}(G,\varphi^*)$ as VS^*.

Fig. 1(a) shows an illustrative example of an undirected graph G with seven vertices and nine edges. Fig. 1(b) depicts a solution (layout) φ of this graph and the *Cut*-value of each position p, $Cut(p,\varphi,G)$. For example, $Cut(1,\varphi,G)=1$ because $L(1,\varphi,G)=\{\mathrm{D}\}$ and $R(1,\varphi,G)=\{\mathrm{A,F,G,E,B,C}\}$ and there is one vertex in L having an adjacent vertex in R. Similarly, $Cut(3,\varphi,G)=2$ where $L(3,\varphi,G)=\{\mathrm{D,A,F}\}$ and $R(3,\varphi,G)=\{\mathrm{G,E,B,C}\}$. The objective function value, computed as the maximum of these cut values, is $\mathrm{VS}(G,\varphi)=3$ whose related position is $p=4$.

The decisional version of the VSP was proved to be NP-complete for general graphs [26]. It is also known that the problem remains NP-complete for planar graphs with maximum degree of three [31], as well as for chordal graphs [15], bipartite graphs [16], grid graphs and unit disk graphs [6].

We can find many different graph problems that, although stated in different terms, are equivalent to the VSP in the sense that a solution to one problem provides a solution to the other one. Some of them are the Path-Width problem [20], the Interval Thickness problem [22], the Node Search Number [23] and the Gate Matrix Layout [21]. The equivalence between these problems is a consequence of the results presented in [13,20,23]. For any graph G let $\mathrm{VS}(G)$, $\mathrm{PW}(G)$, $\mathrm{IT}(G)$, $\mathrm{SN}(G)$ and $\mathrm{GML}(G)$ be the objective function value of the optimal solution for the Vertex Separation, Path-Width, Interval Thickness, Node Search Number and Gate Matrix Layout problems, respectively. These values verify the following relations:

$$\mathrm{VS}(G)=\mathrm{PW}(G)=\mathrm{IT}(G)=\mathrm{SN}(G)-1=\mathrm{GML}(G)+1.$$

The VSP appears in the context of finding "good separators" for graphs [28] where a separator is a set of vertices or edges whose removal separates the graph into disconnected subgraphs. This optimization problem has applications in VLSI design for partitioning circuits into smaller subsystems, with a small number of components on the boundary between the subsystems [25]. The decisional version of the VSP consists of finding a Vertex Separation value larger than a given threshold. It has applications on computer language compiler design and exponential algorithms. In compiler design, the code to be compiled can be represented as a directed acyclic graph (DAG) where the vertices represent the input values to the code as well as the values computed by the operations within the code. An edge from node u to node v in this DAG represents the fact that value u is one of the inputs to operation v. A topological ordering of the vertices of this DAG represents a valid reordering of the code, and the number of registers needed to evaluate the code in a given ordering is precisely the Vertex Separation number of the ordering [4]. The decisional version of VSP has also applications in graph theory [14]. Specifically, if a graph has a Vertex Separation value, say w, then it is possible to find the maximum independent set of G in time $O(2^w n)$. Other practical applications include Graph Drawing and Natural Language Processing [9,29].

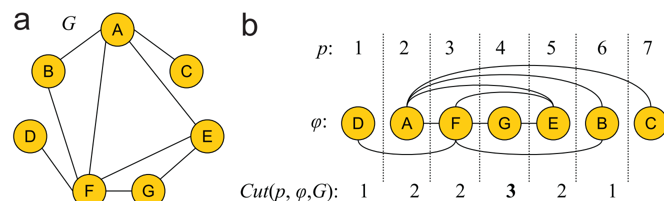

Fig. 1. (a) Graph illustrative example. (b) A layout φ for its VSP.

3. Pure 0-1 optimization model

Before presenting the model, let us define the 0-1 variables that are required:

- x_u^p, whose value is 1 if vertex u is placed in position p in a given feasible layout, i.e. $p=\varphi(u)$ and 0, otherwise (i.e., $p\neq\varphi(u)$), for $u,p=1,2,\ldots,n$.
- $y_{u,v}^{p,q}$, whose value is 1 if $p=\varphi(u)$ and $q=\varphi(v)$ in the given layout φ and 0, otherwise. The inconvenience of this type of variable is that the number of y variables is mn^2 and, then, the cardinality of the set of vertices in graph G must be small in order to obtain the optimal solution of the model by an exact 0-1 solver. On the other hand, $y_{u,v}^{p,q}$ avoids the quadratic expression $x_u^p x_v^q$.
- z_{pc}, whose value is 1 if the vertex in position p (i.e., $p=\varphi(u)$ and, then, $x_u^p=1$) is connected with the vertex in any position, say q (i.e., $q=\varphi(v)$ and, then, $x_v^q=1$) (what implies that $y_{u,v}^{p,q}=1$) that is larger than position c and 0, otherwise.

We propose the following integer programming formulation for the Vertex Separation Problem:

$$\mathrm{VS}^*=\min \mathrm{VS}, \tag{1}$$

subject to

$$\sum_{p\in\{1,\ldots,n\}} x_u^p=1 \quad \forall u=1,\ldots,n, \tag{2}$$

$$\sum_{u\in\{1,\ldots,n\}} x_u^p=1 \quad \forall p=1,\ldots,n, \tag{3}$$

$$y_{u,v}^{p,q}\le x_u^p \quad \forall u,v,p,q=1,\ldots,n, \tag{4}$$

$$y_{u,v}^{p,q}\le x_v^q \quad \forall u,v,p,q=1,\ldots,n, \tag{5}$$

$$x_u^p+x_v^q\le y_{u,v}^{p,q}+1 \quad \forall u,v,p,q=1,\ldots,n, \tag{6}$$

$$z_{pc}\le \sum_{q=c+1}^{n}\sum_{u=1}^{n}\sum_{v=1}^{n} y_{u,v}^{p,q}\le Mz_{pc} \quad \forall p,c=1,\ldots,n-1,\ p\le c, \tag{7}$$

$$\sum_{p=1}^{c} z_{pc}\le \mathrm{VS} \quad \forall c=1,\ldots,n-1, \tag{8}$$

$$x_u^p, y_{u,v}^{p,q}, z_{pc}\in\{0,1\} \quad \forall u,v,p,q=1,\ldots,n. \tag{9}$$

The assignment constraints (2) and (3) ensure that each vertex is only assigned to one position, and each position is only assigned to one vertex, respectively. Subsystems (4)–(6) define the variable $y_{u,v}^{p,q}$ as the product of the variables x_u^p and x_v^q in the traditional way. So, notice that for a position p, $\sum_{q=c+1}^{n} y_{u,v}^{p,q}$ computes the number of edges from the vertex in position p, say $p=\varphi(u)$, to any vertex in a position q, say $q=\varphi(v)$ (since $x_v^q=1$ for $y_{u,v}^{p,q}=1$) that is bigger than c, for $c=q+1,\ldots,n$ for a given layout φ.

Constraints (7) compute the 0-1 value of the z_{pc} variable from the $y_{u,v}^{p,q}$ variables (since $x_u^p=1$ for $y_{u,v}^{p,q}=1$), where M is the standard big-M parameter that should be computationally small enough to allow any feasible layout φ, in our case $M=n-1$.

The left hand side (*lhs*) of constraints (8) gives the number of vertices in positions $\{p\}$ that, on the one hand, are lower than or equal to position c, in any feasible layout φ or equal to c and, on the other hand, they are connected with vertices whose positions are larger than position c. So, there is a position c in layout φ that has the greatest *lhs* in the constraints (8), which is VS^*. The objective function of the model (1) minimizes that value, being VS^* the optimal value.

We can observe that this model has O(mn^2) 0-1 variables and O(mn^3) constraints, what makes it impractical for relatively large instances.

4. Algorithmic approach: variable neighborhood search

In spite of the difficulty of the model presented above 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 an $O(n \log n)$ algorithm for finding the corresponding optimal layout. The algorithm was improved in [36] with a linear time procedure to find the optimal layout. In [33] an alternative method to compute the Vertex Separation of trees was proposed. Ref. [12] proposes an $O(n \log n)$ algorithm to compute the Vertex Separation of unicyclic graphs (i.e., trees with an extra edge). A polynomial-time algorithm to compute the Path-Width (what 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 [12]. Ref. [5] proposes 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 been also 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 [7].

VNS is a metaheuristic for solving optimization problems based on a systematic change of neighborhood structures, without guaranteeing the solution's optimality. In recent years, a large variety of VNS strategies have been proposed. We can highlight the Variable Neighborhood Descent (VND), Reduced VNS (RVNS), Basic VNS (BVNS), Skewed VNS (SVNS), General VNS (GVNS), Variable Neighborhood Decomposition Search (VNDS) and Reactive VNS, among others. We refer the reader to [18] for a complete review of this methodology and its different variants. In this paper, we focus on the Basic VNS variant, see [30] for the details, which combines deterministic and stochastic changes of neighborhood as shown in the BVNS pseudo-code depicted in Algorithm 1.

Algorithm 1. BasicVNS ($k_{\max}$, $t_{\max}$).

1. $\varphi \leftarrow Construct()$
2. **repeat**
3. $k \leftarrow 1$
4. **repeat**
5. $\varphi' \leftarrow Shake(\varphi, k)$
6. $\varphi'' \leftarrow LocalSearch(\varphi')$
7. $NeighborhoodChange(\varphi, \varphi'', k)$
8. **until** $(k = k_{max})$
9. $t \leftarrow Time()$
10. **until** $(t = t_{max})$

The method starts by constructing a feasible solution (step 1), using one of the constructive procedures described in Section 4.1. The BVNS implementation is executed for a predefined computing time, $t_{\max}$ (steps 2–10). The search process starts with the first neighborhood of the constructed solution, $N_1(\varphi)$ (step 3). Then, BVNS performs stochastic changes of neighborhood structures until reaching the largest predefined neighborhood $k_{\max}$ (steps 4–8). VNS has three main strategies within the main loop, namely, shaking (step 5), improvement (step 6) and neighborhood change (step 7). In the shaking stage a solution, say φ', is generated within $N_k(\varphi)$, where $k = 1, \ldots, k_{\max}$ identifies a neighborhood structure within a set of predefined neighborhoods (see Section 4.3 for additional details). Then, the improvement method (Section 4.5) is applied to φ' in order to find a local optimum, say φ'', in the corresponding neighborhood. Finally, the neighborhood change stage (Section 4.4), analyzes if φ'' is better than φ (see in Section 4.5 the extended definition of the concept of improvement move). If so, φ is replaced with φ'' and k is set to one. Otherwise, k is incremented by one unit. And, in any case, we repeat the procedure.

We now describe with more detail the main strategies of our BVNS approach for solving the Vertex Separation Problem.

4.1. Constructive procedures

We have designed two greedy constructive algorithms, say C1 and C2, for the VSP. The constructive procedure, C1, starts by creating a set of unlabeled vertices U (initially $U = V$), and a set of labeled vertices $L = V \backslash U$. The vertex with the minimum degree is selected as the first node u (ties are broken at random). The vertex u is labeled with 1. Then, sets U and L are properly updated (i.e., $U = U \backslash \{u\}$ and $L = L \cup \{u\}$). Once the first label is assigned to vertex u, C1 evaluates the greedy function as follows:

$$g(v) = |N_L(v)| - |N_U(v)| \quad \forall v \in U,$$

where $N_L(v)$ is the set of vertices adjacent to v that has been already labeled, and $N_U(v)$ is the set of vertices adjacent to v not labeled yet. The constructive procedure selects the vertex with the maximum g-value and assigns the next label to it. The procedure ends when all the vertices of the graph have a label (i.e., $U = \emptyset$).

The second constructive procedure, C2, is based on the creation of the so-called level structures [27] which means that the set of vertices V is partitioned into different sets $L_1, L_2, \ldots, L_\lambda$ called levels. The first level, L_1, contains only one vertex. The rest of levels L_l with $l = 2, 3, \ldots, \lambda$ (where λ indicates the number of levels) contains all the vertices adjacent to some vertex in L_{l-1} that are not present in any L_j with $1 \le j < l$. The number of levels λ exclusively depends on the graph and the vertex placed in L_1.

The level structure created in this way guarantees that the vertices in alternative levels are not adjacent. In order to construct this level structure we use a breadth first search approach, performing the search once for each vertex of the graph. Therefore, if the graph has n vertices, we construct n different level structures. In the context of the VSP, it is desirable to obtain a level structure with λ as large as possible (i.e., a structure with the largest number of levels). Once we have identified the most suitable level structure, C2 explores it performing a breadth-first search and assigning levels in an incremental way, thus obtaining a solution for the VSP.

4.2. Updating the objective function

Let φ_p be a partial solution where p vertices have been placed in positions from 1 to p. For the sake of simplicity, we denote v_p as the vertex placed in position p, i.e., $p = \varphi(v_p)$. We maintain this notation for the rest of the paper. Given a graph G and a partial solution φ_p, the Cut-value of a position p, $Cut(p, \varphi, G)$ can be computed from the Cut-value of the previous position $p-1$ (except for $p = 1$ where, obviously, the Cut-value is always 1) as follows,

$$Cut(p, \varphi_p, G) = Cut(p-1, \varphi_p, G) + \delta^+(p) - \delta^-(p),$$

where $\delta^+(p) \in \{0, 1\}$ has the value 1 if vertex v_p has, at least, one adjacent vertex in $N_U(v_p)$, and 0, otherwise; and $\delta^-(p) = |N_L(v_p)|$.

This strategy can be extended to the incremental computation in complete solutions (i.e., solutions $\varphi = \varphi_r$ with $r = n$). Specifically,

for any position p the set $N_U(v_p)$ is replaced with the set of vertices placed in a position q with $1 \le q \le p$. Symmetrically, the set $N_L(v_p)$ is replaced with the set of vertices placed in a position q with $p < q \le n$. Fig. 2 shows an example of the incremental objective function computation. Let us consider the third position in φ (i.e., vertex C). The *Cut*-value of this position can be computed using the *Cut*-value of the previous position, $Cut(2,\varphi,G) = 2$. The value of $\delta^+(3)$ is 1 because vertex C has one adjacent at least (specifically, three adjacents) placed in a position greater than $\varphi(C) = 3$ (vertices B, F and E, with $\varphi(B) = 4$, $\varphi(F) = 5$ and $\varphi(E) = 6$, respectively). On the other hand, the value of $\delta^-(3)$ is 2 because C is the adjacent vertex with the largest label of vertices A and D, placed in positions $\varphi(A) = 1$ and $\varphi(D) = 2$, respectively. The same reasoning can be applied to compute the rest of the *Cut*-values.

4.3. Shake

In this section we propose a shake function for the VSP, called *shake*(φ,k). This procedure initially selects the vertices to be moved, based on their *Cut*-value. Specifically, *shake*(φ,k) selects the k vertices in V with the largest *Cut*-value in φ. Then, each selected vertex is exchanged with another vertex determined at random, obtaining a new ordering φ'. The rationale behind this procedure is that the k selected vertices have *Cut*-values close or equal to the maximum *Cut*-value for φ in G and, therefore, the aim is to reduce their *Cut*-value improving the objective function of φ'. Additionally, it is expected that the shaking step will contribute to the diversification of the search process. In other words, shaking will produce a solution "far away" from the current one, allowing the search to explore different regions of the solution space.

4.4. Neighborhood structures

A solution to the VSP can be represented as an ordering, where each vertex is located in the position given by its label. For example, the labeling of the graph depicted in Fig. 1 can be expressed by the ordering $\varphi = (D,A,F,G,E,B,C)$, where the first vertex, D, in the ordering receives label 1, the second vertex, A, receives label 2, and so on. We define neighborhood structures based on the exchange of vertices in a given ordering. Given a solution $\varphi = (v_1, \ldots, v_p, \ldots, v_q, \ldots, v_n)$, we define *Move*$(\varphi,p,q)$ as exchanging in φ the vertex in position p (i.e., v_p) with the vertex in position q (i.e., v_q) and, thus, producing a new solution $\varphi' = (v_1, \ldots, v_q, \ldots, v_p, \ldots, v_n)$.

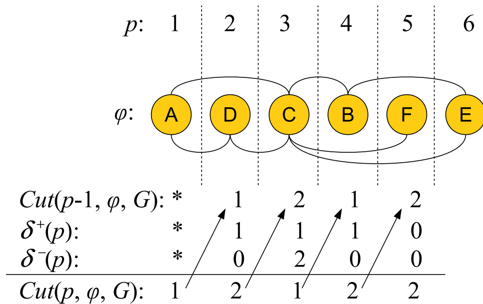

Fig. 2. Incremental objective function computation. The symbol * means that the corresponding value is not defined for the first position.

In a direct implementation, the complexity of evaluating φ' is $O(n^2)$ in the worst case since it requires to visit, for each position p in φ, all the vertices with labels $p \le q$. However, the *Cut*-value of some vertices does not change when we perform a move. Therefore, it is not required to compute them again. Specifically, if we perform *Move*(φ,p,q), all vertices with label r for $1 \le r < p$ or $q \le r \le n$ do not change their *Cut*-values. As a consequence, we only need to update vertices whose label s is such that $p \le s < q$. For example, Fig. 3(a) shows a layout and Fig. 3(b) represents the layout resulting from performing *Move*$(\varphi,2,5)$. This move only affects the *Cut*-value of positions $2 \le s < 5$ (represented as a highlighted band in those figures).

Additionally, to compute the *Cut*-value of the vertices involved (i.e., the set of vertices {D,C,B} in Fig. 3(b)) we can use the incremental objective function computation described above, but restricted to vertices in positions from p to $q-1$. In the example shown in Fig. 3(b) we can observe that the update of the *Cut*-values is only needed for the vertices in positions $\varphi(D) = 2$, $\varphi(C) = 3$ and $\varphi(B) = 4$.

Given a layout φ, its neighborhood $N_1(\varphi)$ is defined as all possible exchanges between each pair of vertices. In other words, a solution φ' belongs to $N_1(\varphi)$ if and only if φ and φ' only differ in two labels. In general, we may say that a solution φ' belongs to the kth neighborhood of solution φ (i.e., $\varphi' \in N_k(\varphi)$) if φ and φ' differ in $k+1$ labels.

4.5. Local search

VSP is a min–max problem [8,32,35] where the value of the objective function is usually reached in several positions of the permutation φ. This kind of problems present a "flat landscape", which turns out in a challenge for classical local search procedures as well as for 0-1 solvers to obtain the optimal layout. 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 p of a particular vertex v_p in φ (i.e., obtaining a new solution φ') such that its *Cut*-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 *Cut*-value is reduced, regardless whether the objective function improves or not. Considering this extended definition of "improving" we overcome the lack of information provided by the objective function. Specifically, we implement a candidate list strategy classifying the vertices of the graph according to its *Cut*-value. For example, given the graph depicted in Fig. 1 and the labeling φ, we obtain three different sets: $S_3(\varphi) = \{G\}$, containing vertices with *Cut*-value equal

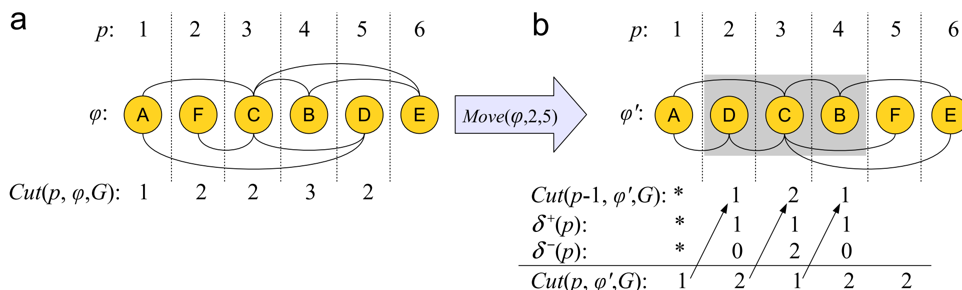

Fig. 3. Illustrative example of an interchange move: (a) layout before the move and (b) layout after the move.

to 3; $S_2(\varphi)=\{$A,F,E$\}$, with *Cut*-value equal to 2; and finally $S_1(\varphi)=\{$D,B$\}$ with *Cut*-value equal to 1.

Given the definition of $S_i(\varphi)$ that has been introduced in the previous paragraph, we consider that a move improves the current solution if any node involved in the move is removed from $S_i(\varphi)$ and included in $S_j(\varphi)$ with $j<i$ without increasing the cardinality of any set $S_l(\varphi)$ for $l>i$. According to this definition of improving, a move which removes a vertex from $S_2(\varphi)$ (for example, vertex A) including it in $S_1(\varphi)$ is considered an improving move if the cardinality of $S_3(\varphi)$ remains unaltered. We have empirically found that this criterion allows the local search procedure to explore a larger number of solutions than a typical implementation that only performs moves when the objective function is improved. Fig. 4(a) shows an improving move for the labeling of Fig. 3(a). The move consists of exchanging the labels of vertices C and A and, thus, obtaining a new solution φ'. In the new labeling, vertex F is removed from set $S_2(\varphi)$ and included in set $S_1(\varphi')$. It means that the *Cut*-value of vertex F is decreased by 1 unit. This move does not reduce the VS value of the graph, but it reduces the number of vertices with large *Cut*-value. Observe that this move does not improve the incumbent solution, but it can allow further moves that, at the end, improves it. On the other hand, if we now exchange the labels of vertices B and C obtaining a new solution φ'' (see Fig. 4(b)) then vertex C is removed from set $S_2(\varphi)$ and included in set $S_3(\varphi'')$ (i.e., its *Cut*-value is increased by one unit). Therefore, although this move does not affect the VS value, it is not accepted. Algorithm 2 shows a pseudocode of the procedure, where the acceptance criterion has been implemented. This procedure starts by comparing the objective function of the previous ordering (φ) and the related function of the ordering after the corresponding movement (φ'). If the associated move reduces the value of the objective function, then *IsImprovementMove* returns *true* (step 2). On the other hand, if the move deteriorates the objective function value then this procedure returns *false* (step 4). In case that the corresponding move does not affect the value of the objective function (steps 6–13) *IsImprovementMove* considers the extended definition of improvement move defined above.

Algorithm 2. *IsImprovementMove*(φ,φ').

1. **if** $VS(G,\varphi') < VS(G,\varphi)$ **then**
2. **return** *true*
3. **else if** $VS(G,\varphi') > VS(G,\varphi)$ **then**
4. **return** *false*
5. **else** [$*VS(G,\varphi)=VS(G,\varphi')*$]
6. **for** $i \leftarrow VS(G,\varphi)$ **downto** 1 **do**
7. **if** $|S_i(\varphi)| > |S_i(\varphi')|$ **then**
8. **return** *true*
9. **else if** $|S_i(\varphi)| < |S_i(\varphi')|$ **then**
10. **return** *false*
11. **endif**
12. **end for**
13. **return** *false*
14. **end if**

The proposed local search strategy is presented in Algorithm 3. This method receives a layout φ and perform moves while an improve is produced (i.e., while-loop in step 2). Specifically, the procedure starts by arranging the vertices in descending order of the *Cut*-value, obtaining the set *posSet* which contains the position of such vertices (step 4). Then, the algorithm traverses this set (for-loop in step 5) trying to interchange each vertex in *posSet* with the remaining vertices in φ (for-loop in step 6). In each iteration, the method proves to interchange the position of the two considered vertices (step 8) accepting the move if the new layout φ' outperforms φ (steps 9–12). This procedure is repeated until no improvement in the move is found.

Algorithm 3. *LocalSearch*(φ).

1. *improvement* $\leftarrow$ *true*
2. **while** *improvement* **do**
3. *improvement* $\leftarrow$ *false*
4. *posSet* $\leftarrow$ *OrderByCutValue*(φ)
5. **for all** $p \in posSet$ **do**
6. **for** $q \leftarrow 1$ **to** $|\varphi|$ **do**
7. **if** $p \neq q$ **then**
8. $\varphi' \leftarrow$ *Interchange*(φ,p,q)
9. **if** *IsImprovementMove*(φ,φ') **then**
10. $\varphi \leftarrow \varphi'$
11. *improvement* $\leftarrow$ *true*
12. **end if**
13. **end if**
14. **end for**
15. **end for**
16. **end while**
17. **return** φ

5. Computational experience

This section reports the computational experiments that we have performed for testing the efficiency of our VNS procedure, so-called BVNS, for solving the VSP. The algorithm has been implemented in Java SE 6 and all the experiments were conducted on an Intel Core i7 2600 CPU (3.4 GHz) and 2 Gb RAM. We have experimented with three sets of instances, totalizing 173 instances (all of the instances are available at http://www.optsicom.es/vsp/).

5.1. Testbed description

- HB: We derived 73 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 73 graphs with $n \leq 1000$. The number of vertices and edges range 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 [34]. They are also called two

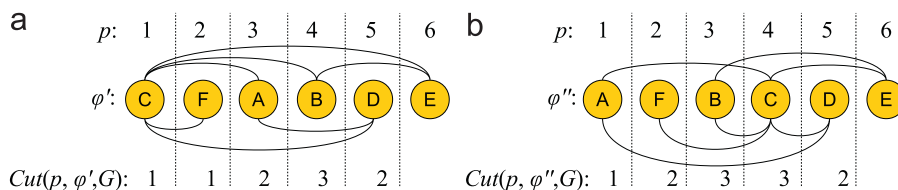

Fig. 4. (a) Improving move and (b) non-improving move over the layout depicted in Fig. 3(a).

dimensional meshes and the optimal solution of the VSP for squared grids is known by construction, see [7]. 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 ranges 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 [11], 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 [11] 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 ranges from 22 to 202 and from 21 to 201, respectively.

Eight experiments are performed for assessing the validation of the proposed procedure BVNS. We have selected 32 `HB` representative instances, with different sizes and densities, to perform the experiments 1–5 oriented to establish the best configuration of the BVNS procedure. Let us name this set of instances the `HB` subset. Specifically, we consider the instances `ASH85`, `BCSPWR01`, `BCSPWR02`, `BCSPWR03`, `BCSSTK01`, `BCSSTK02`, `BCSSTK03`, `BCSSTK04`, `BCSSTK05`, `BCSSTK22`, `CAN_144`, `CAN_161`, `CAN_187`, `CAN_229`, `CAN_24`, `CAN_61`, `CAN_62`, `CAN_73`, `CAN_96`, `DWT_162`, `DWT_193`, `DWT_198`, `DWT_209`, `DWT_221`, `DWT_234`, `DWT_245`, `DWT_59`, `DWT_66`, `DWT_72`, `DWT_87`, `NOS1` and `NOS4`. Experiment 6 evaluates the performance of the optimization model (1)–(9). Experiment 7 evaluates the performance of BVNS on instances whose optimum is known and, finally, the performance of the best configuration of the procedure BVNS is analyzed using the full testbed of 173 instances (experiment 8).

5.2. Experiment 1: constructive procedures performance

In our first experiment we compare the performance of the two proposed constructive procedures for the VSP, namely C1 and C2, described in Section 4.1. We have conducted the experiment over the `HB` subset of 32 instances presented above. We generate one solution with each constructive procedure. The statistics of Table 1 are as follows: # 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; time, average computing time in seconds required by the procedure.

Table 1 shows that C1 obtains better results than C2 in all headings and using similar computing time. Specifically, C1 reaches a smaller deviation (11.24%) than C2 (15.66%) and a larger number of best solutions (19 versus 18) in the set of instances experimented with.

Table 1
Constructive procedures.

	C1	C2
# best	19	18
Avg.	17.50	17.65
Dev (%)	11.24	15.66
Time	0.010	0.018

5.3. Experiment 2: local search performance

We compare the two constructive procedures for the VSP (C1 and C2) coupled with the improvement procedure (LS) presented in Section 4.5. Table 2 reports the results using the same headings as above. We can observe that the method C2 coupled with LS gets more best solutions than C1+LS (22 versus 19), smaller deviation (9.48% versus 9.86%) and requires 7% smaller computing time (26.25 s versus 24.42 s) than the method C1 + LS. Despite C1 alone has better performance than C2 (see Table 1), when coupling the improvement strategy to the constructive procedures the behavior completely changes and C2 + LS clearly outperforms C1 + LS. This behavior may be partially explained by the fact that C2 is able to construct layouts with broader diversity than C1.

Table 2
Constructive procedures coupled with the improvement procedure.

	C1 + LS	C2 + LS
# best	19	22
Avg.	14.88	14.88
Dev (%)	9.86	9.48
Time	26.25	24.42

5.4. Experiment 3: objective function computation

We study the computing time of C2 + LS when (1) using a direct objective function computation (DOFC) and (2) using the incremental objective function computation (IOFC) presented in Section 4.2. Fig. 5 depicts a bar diagram where the X -axis represents the ten largest instances of the `HB` subset and the Y -axis gives the computing time required to obtain a local optimum for both methods, respectively, (DOFC and IOFC) in the corresponding instance. The figure clearly shows that the saving in computing time is significant for IOFC. Specifically, for these ten instances DOFC needs 69 s on average to obtain a local optimum, while IOFC requires 3.85 s on average to obtain the same optimum value, i.e., almost 18 times faster.

5.5. Experiment 4: number of neighborhoods

This experiment consists of evaluating the impact of the k_{max} parameter on the performance of BVNS. It is expected that the larger the value of k_{max}, the larger the computing time and, hopefully, the lower the objective function value. Table 3 shows the number of best solutions, the average objective function, the average deviation and the computing time for the four different values of $k_{max} = \{0.08n, 0.1n, 0.3n, 0.5n\}$, where n indicates the number of vertices. The table shows that $k_{max} = 0.5n$ obtains the best results in terms of quality but using the highest computing time. On the other hand, $k_{max} = 0.08n$ is the fastest BVNS variant but it has a poor quality results. Then, we select $k_{max} = 0.3n$ as a trade-off between quality and computing time.

5.6. Experiment 5: one-start versus multi-start strategy

In general, the shake strategy allows VNS to escape from local optima. Additionally, this procedure also diversifies the incumbent solution by incorporating/removing elements in the shaken one. In other words, the shake algorithm produces a solution which moves away from the current incumbent solution. However, it could be possible that the shake strategy is not powerful enough to be used alone (i.e., deep local optimum). Consequently,

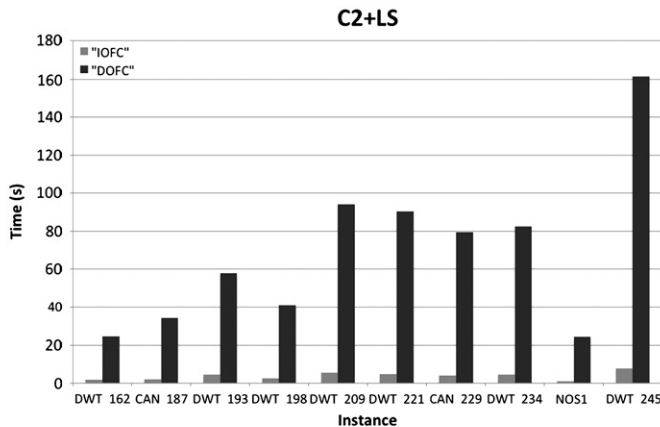

Fig. 5. Computing time comparison for direct (DOFC) and incremental (IOFC) objective function computation.

Table 3
Comparison of different $k_{\max}$ values.

	$k_{\max}$			
	$0.08n$	$0.1n$	$0.3n$	$0.5n$
# best	27	28	31	32
Avg.	13.34	13.31	12.91	12.87
Dev (%)	3.71	2.92	0.78	0.00
Time	69.70	79.54	214.32	341.74

in order to analyze the potential improvement of diversification, we consider a multi-start strategy which construct solutions in different positions of the search space and, then, execute the VNS. In this experiment we compare the performance of the best identified variant of BVNS (which includes C2, exchange-based LS and the parameter $k_{\max}$ set to $0.3n$) with a multi-start BVNS. In order to have a fair comparison, both methods use the same constructive and search strategies. The multi-start BVNS is executed starting from 10 different solutions. So, it is expected that its computing time is about 10 times slower, and the $k_{\max}$ parameter is set to $0.03n$, so it is 10 times lower than the $k_{\max}$ value in BVNS. Table 4 shows that the best variant of BVNS while using the one-start approach has better performance than the multi-start strategy in the same environment. Specifically, the former has more best solutions (30 versus 25) and it reaches smaller average deviation (0.40%) than the latter (4.46%) in the set of 32 instances of the HB subset considered in the experimentation. Furthermore, the computing time for the one-start BVNS (214.32 s) is a 93% less than the time required by the multi-start strategy (3170.09 s) for BVNS. It is important to remark that in both variants, the BVNS algorithm is exactly the same.

5.7. Experiment 6: pure 0-1 optimization model

This experiment consists of evaluating the optimization of the mathematical model introduced in Section 3, using the state-of-the-art optimization engine CPLEX v12.1 [19].

The main results of the experiment are shown in Table 5, whose headings indicate the name of the instance (*Instance*), number of vertices (n), number of constraints (n_c), number of 0-1 variables (n_{01}), number of constraint matrix non-zero elements (n_{el}), number of CPLEX branch-and-cut required for obtaining the incumbent Vertex Separation (nn), optimal Vertex Separation value for the corresponding instance G ($VS^* = VS(G)$), LP lower bound of the optimal Vertex Separation value provided by CPLEX ($\underline{VS}_{CPLEX}$),

Table 4
One-start versus multi-start BVNS.

	BVNS	Multi-start BVNS
# best	30	25
Avg.	12.91	13.41
Dev (%)	0.40	4.46
Time	214.32	3170.09

optimality gap in percentage ($GAP = (VS^* - \underline{VS}_{CPLEX})/\underline{VS}_{CPLEX}$), Vertex Separation value related to the incumbent Vertex Separation provided by CPLEX, computing time (s) required by CPLEX ($Time_{CPLEX}$), Vertex Separation value provided by our metaheuristic procedure BVNS (VS_{BVNS}), and computing time (s) required by the procedure ($Time_{BVNS}$).

Table 5 reports the main results of the following small instances: two instances from the grid set and the 21 instances from the set that have 22 nodes. In total 23 out of 173 instances in the sets that we have experimented with. We selected those instances where CPLEX could be used for its optimization. It is important to remark that CPLEX could not execute the model for the rest of instances since there was not enough memory (in our computer) to allocate the model, due to its large dimensions. Notice that the optimal Vertex Separation of the subset of instances in Table 5 is known by construction.

We can observe in Table 5 that the LP lower bound $\underline{VS}_{CPLEX}$ is so small that is useless, in fact is 1, i.e., the minimum possible value for connected graphs. Additionally, we can observe that CPLEX only obtains the optimal Vertex Separation for the smallest instance, grid_3. On the other hand, the difference between the CPLEX incumbent solution and the optimum (VS^*) is one or two units. Attending to the optimality gap, we can observe the weakness of the 0-1 model. Finally, it is worth to mention that, although CPLEX obtains the optimal value $VS^* = 3$ for instance Tree_22_3_rot4, it can not guarantee its optimality in the computing time limit, 5400 s (taking into account the value of the lower bound $\underline{VS}_{CPLEX} = 1 < VS^* = 3$).

Finally, we can observe in Table 5 that the procedure BVNS obtains the optimal Vertex Separation value in all instances that we have used in the experiment, being impressive its computing time (much less than 0.15 s for each instance). On the other hand, since our BVNS is heuristic in nature, it does not guarantee the optimality of these solutions by itself.

5.8. Experiment 7: BVNS on instances with known optimum value

We evaluate in this experiment the performance of our best identified variant of BVNS on a larger set of instances than the set used in the previous experiment. In this other set the optimal Vertex Separation is known by construction. Specifically, we use the 50 instances of the set Grids and 50 instances of the set Trees and test whether the BVNS is able to match the optimum or not. Table 6 has the same headings as the other tables plus the computing time, say, Time_to_opt, required by BVNS to reach the optimal solution on average.

Observe that BVNS is able to find the optimum solution in all the instances of the set Grids, subset Trees_22 and subset Trees_67. Therefore, BVNS is able to find the optimum in the instances where CPLEX could not due to running out of memory while solving the model presented in Section 3. Indeed, BVNS finds the optimum in subsets Trees_22 and Trees_67 in short computing time. It is worth to remark that although the BVNS method requires a relative large computing time (1422.76 s) for the whole procedure, it obtains the optimal solution in only 0.36 s on average.

Table 5
CPLEX results. Model dimensions, optimal solution, CPLEX incumbent solution, BVNS solution and computing times (s).

Instance	*n*	n_c	n_{01}	n_{el}	*nn*	VS^*	$\underline{VS}_{CPLEX}$	*GAP* (%)	VS_{CPLEX}	$Time_{CPLEX}$	VS_{BVNS}	$Time_{BVNS}$
grid_3	9	5938	2062	19,690	12,636	3	3	0.00	3	2528	3	0.054
grid_4	16	37,166	12,665	152,318	902	4	1	300.00	5	5400	4	0.126
Tree_22_3_rot0_3	22	61,532	21,044	2,922,994	533	3	1	200.00	4	5400	3	0.029
Tree_22_3_rot1_3	22	61,532	21,044	2,922,994	543	3	1	200.00	5	5400	3	0.022
Tree_22_3_rot2_3	22	61,532	21,044	2,922,994	604	3	1	200.00	4	5400	3	0.024
Tree_22_3_rot3_3	22	61,532	21,044	2,922,994	557	3	1	200.00	4	5400	3	0.037
Tree_22_3_rot4_3	22	61,532	21,044	2,922,994	553	3	1	200.00	3	5400	3	0.019
Tree_22_3_rot5_3	22	61,532	21,044	2,922,994	572	3	1	200.00	4	5400	3	0.018
Tree_22_3_rot6_3	22	61,532	21,044	2,922,994	593	3	1	200.00	4	5400	3	0.015
Tree_22_3_rot0_1	22	61,532	21,044	2,922,994	595	3	1	200.00	4	5400	3	0.04
Tree_22_3_rot1_1	22	61,532	21,044	2,922,994	597	3	1	200.00	4	5400	3	0.02
Tree_22_3_rot2_1	22	61,532	21,044	2,922,994	572	3	1	200.00	4	5400	3	0.018
Tree_22_3_rot3_1	22	61,532	21,044	2,922,994	577	3	1	200.00	4	5400	3	0.025
Tree_22_3_rot4_1	22	61,532	21,044	2,922,994	590	3	1	200.00	4	5400	3	0.016
Tree_22_3_rot5_1	22	61,532	21,044	2,922,994	566	3	1	200.00	4	5400	3	0.019
Tree_22_3_rot6_1	22	61,532	21,044	2,922,994	646	3	1	200.00	5	5400	3	0.022
Tree_22_3_rot0_2	22	61,532	21,044	2,922,994	625	3	1	200.00	5	5400	3	0.029
Tree_22_3_rot1_2	22	61,532	21,044	2,922,994	590	3	1	200.00	5	5400	3	0.022
Tree_22_3_rot2_2	22	61,532	21,044	2,922,994	547	3	1	200.00	4	5400	3	0.024
Tree_22_3_rot3_2	22	61,532	21,044	2,922,994	588	3	1	200.00	4	5400	3	0.037
Tree_22_3_rot4_2	22	61,532	21,044	2,922,994	591	3	1	200.00	4	5400	3	0.019
Tree_22_3_rot5_2	22	61,532	21,044	2,922,994	543	3	1	200.00	4	5400	3	0.018
Tree_22_3_rot6_2	22	61,532	21,044	2,922,994	636	3	1	200.00	4	5400	3	0.015

Note: CPLEX computing time limit: 5400 s.

Table 6
Grids and trees.

	Grids	Trees_22	Trees_67	Trees_202
# opt	50	15	15	1
Avg.	29.5	3.00	4.00	6.35
Dev (%)	0.00	0.00	0.00	27.00
Time_to_opt	0.36	0.002	0.58	*
Time	1422.76	0.02	3.37	314.28

Table 7
Constructive versus constructive plus local search versus BVNS.

	C2	C2+LS	BVNS
# best	16	38	73
Avg.	30.70	26.68	24.25
Dev (%)	35.97	11.73	0.00
Time	1417.05	1204.38	749.77

In sight of these results we can conclude that grids instances are easily solved by our procedure. Regarding the instances of the set Trees the algorithm is able to find the optimal solution for the small and medium instances (i.e., Trees_22 and Trees_67) in very small computing time, 0.002 s for finding the solution and 0.02 s for proving it for trees with 22 nodes, and 0.58 s and 3.37, respectively, for trees with 67 nodes. Specifically, BVNS only finds the optimal solution in one out of 20 large instances included in the set Trees_202 and the average deviation is 27.0%. But, although the computing time, 314.28 s, is large as compared to the other sets of instances, it is not too much effort considering that the instances have 202 vertices. Moreover, although it is not reported in Table 6, we can observe analyzing each instance that, besides guaranteeing optimally in one instance, BVNS obtains in 11 instances a quasi-optimal solution (the solution value is 6, i.e., one unit from the optimal one), and for the remaining 8 instances it obtains a solution value of 7 (i.e., two units from the optimal one). Notice that, since BVNS does not match the optimal value in 19 instances, the corresponding Time_to_opt value is not reported (represented by an asterisk in the table).

5.9. Experiment 8: BVNS procedure versus C2 and C2 + LS

The VSP has been computationally observed to be optimally solved for particular graphs (trees, grids, co-graphs, etc.). Additionally, as it was presented in Section 2, VSP has relevant practical applications. However, as far as we know, there are no previous heuristic procedures to obtain high-quality solutions on general graphs. In order to provide a final comparison, our experiment consists of evaluating the performance of BVNS versus the constructive approach C2, and the constructive C2 plus the local search procedure, C2 + LS, executed as stand-alone procedures. To provide a fair comparison, the three procedures are executed during a similar computing time. Specifically, C2 is run for 23,000 independent iterations while C2 + LS is executed for 150 independent iterations. The target of this computational comparison is to experimentally show how a systematic change of neighborhood (VNS) is able to outperform more direct approaches (say C2 and C2 + LS).

Finally, the results of the last experiment are shown in Table 7, where the performance of the procedures C2, C2 + LS and BVNS are compared by executing them over the whole set of 73 HB instances, where the optimum is not known. Taking into account that we are considering large instances (up to 1000 vertices), the computing time could not be eventually affordable. Therefore, the time limit for stopping the heuristic methods was set to 1000 s. We can observe that the inclusion of the local search procedure LS described in Section 4.5 improves the solution obtained by the constructive approach C2 alone. Furthermore, BVNS clearly outperforms C2 and C2 + LS in all headings. Specifically, BVNS is able to find the best solution in the 73 instances under consideration (in 749.77 s), while C2 + LS finds it in only 38 instances (requiring 1204.38 s) and C2 only finds it in 16 instances (requiring 1417.05 s).

6. Conclusions

In this paper we have proposed a pure 0-1 optimization model for the Vertex Separation Problem (VSP). This model has $O(mn^2)$ variables and $O(mn^3)$ constraints, which makes it impractical for relatively large instances, as we have reported in our experimentation. We therefore have also presented an approximation algorithm. As far as we know, it is the first heuristic procedure for general graphs for the VSP. Specifically, we have introduced two constructive procedures based on different greedy strategies, so-called C1 and C2. Experimental results show that C1 marginally improves the performance of C2. Additionally, we introduce a novel scheme for calculating the objective function which substantially reduces the computing time (by a factor of 18) as compared with the direct implementation. We also propose a local search strategy, LS, based on exchanges that incorporates a new definition of the improvement move. It allows the procedure to search in flat landscapes. Experimental results revealed that coupling constructives procedures with the local search strategy, improves the results of constructive methods by themselves in terms of quality although consuming more computing time. We can observe in our extensive experimentation that the best combination is C2 + LS which means that although as a constructive procedure C1 obtains better results than C2, when combined with the local search C2 becomes the best alternative. We also introduce a shake procedure which selects vertices according to their contribution, performing an interchange in a stochastic way (in order to favor diversification). Finally, we incorporate C2, LS and the shake procedure in a BVNS algorithm. The proposed metaheuristic has been tested on a large benchmark consisting of 173 well-known instances in the existing literature. Specifically, the BVNS has been able to find the optimal Vertex Separation in 81 out of 100 instances (whose optimal solution is known by construction). In the rest of instances, where the optimum value is unknown, the BVNS procedure clearly outperforms C2 and C2 + LS when executing all the procedures with the same running time proving experimentally the superiority of the BVNS metaheuristic.

Acknowledgments

This research has been partially supported by the Spanish Ministry of Science and Innovation, grants TIN2009-07516 and TIN2011-28151, and the Government of the Community of Madrid, grant 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 Journal of Discrete Mathematics 1995;8(4):606–616.
[3] Bodlaender HL, Gilbert JR, Hafsteinsson H, Kloks T. Approximating treewidth, pathwidth, frontsize, and shortest elimination tree. Journal of Algorithms 1995;18(2):238–255.
[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. 574–3.
[5] Bollobás B, Leader I. Edge-isoperimetric inequalities in the grid. Combinatorica 1991;11:299–314.
[6] Díaz J, Penrose MD, Petit J, Serna M. Approximating layout problems on random geometric graphs. Algorithms 2001;39(1):78–116.
[7] Díaz J, Petit J, Serna M. A survey of graph layout problems. ACM Computing Survey 2002;34(3):313–356.
[8] Duarte A, Martí R, Resende MGC, Silva RMA. GRASP with path relinking heuristics for the antibandwidth problem. Networks 2011;58(3):171–189.
[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):267–292.
[10] Ellis JA, Sudborough IH, Turner JS. The vertex separation and search number of a graph. Information and Computation 1978;113:50–79.
[11] Ellis JA, Sudborough IH, Turner JS. The vertex separation and search number of a graph. Journal of Information and Computation 1994;113:50–79.
[12] Ellis JA, Markov M. Computing the vertex separation of unicyclic graphs. Information Computing 2004;192(2):123–161.
[13] Fellows MR, Langston MA. On search, decision and the efficiency of polynomial-time algorithms. Journal of Computing Systems Sciences 1994;49(3): 769–779.
[14] Fomin FV, Hie K. Pathwidth of cubic graphs and exact algorithms. Information Processing Letters 2006;97:191–196.
[15] Gustedt J. On the pathwidth of chordal graphs. Discrete Applied Mathematics 1993;45(3):233–248.
[16] Goldberg PW, Golumbic MC, Kaplan H, Shamir R. Four strikes against physical mapping of DNA. Journal of Computing Biology 1995;12(1):139–152.
[18] Hansen P, Mladenovic N, Moreno JA. Variable neighbourhood search: methods and applications. Annals of Operations Research 2010;175(1):367–407.
[19] IBM ILOG CPLEX, User manual; 2009.
[20] Kinnersley NG. The vertex separation number of a graph equals its pathwidth. Information Processing Letters 1992;42(6):345–350.
[21] Kinnersley NG, Langston MA. Obstruction set isolation for the gate matrix layout problem. Discrete Applied Mathematics 1994;54(2–3):169–213.
[22] Kirousis M, Papadimitriou CH. Interval graphs and searching. Discrete Mathematics 1985;55(2):181–184.
[23] Kirousis M, Papadimitriou CH. Searching and pebbling. Theory of Computation Sciences 1986;47(2):205–218.
[25] Leiserson CE. Area-efficient graph layouts (for VLSI). In: Proceedings of IEEE symposium on foundations of computer science; 1980. p. 270–1.
[26] Lengauer T. Black–white pebbles and graph separation. Acta Informatica 1981;16:465–475.
[27] Lewis JG. The Gibbs–Poole–Stockmeyer and Gibbs–King algorithms for reordering sparse matrices. ACM Transactions on Mathematical Software 1982;8:190–194.
[28] Lipton RJ, Tarjan RE. A separator theorem for planar graphs. SIAM Journal of Applied Mathematics 1979;36:177–189.
[29] Miller GA. The magical number seven, plus or minus two. SIAM Journal of Applied Mathematics 1956;13:81–97.
[30] Mladenović N, Hansen P. Variable neighborhood search. Computers and Operations Research 1997;24:1097–1100.
[31] Monien B, Sudborough IH. Min cut is NP-complete for edge weighted trees. Theory of Computation Sciences 1988;58:209–229.
[32] Pantrigo JJ, Martí R, Duarte A, Pardo EG. Scatter search for the cutwidth minimization problem. Annals of Operations Research, doi:10.1007/s10479-011-0907-2.
[33] 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 fourth annual international conference on computing and combinatorics, COCOON'98; 1998. p. 279–8.
[34] Raspaud A, Schröder H, Sýkora O, Török L, Vrt'o I. Antibandwidth and cyclic antibandwidth of meshes and hypercubes. Discrete Mathematics 2009;309: 3541–3552.
[35] Resende MGC, Martí R, Gallego M. Duarte a GRASP and path relinking for the max-min diversity problem. Computers and Operations Research 2010;37: 498–508.
[36] Skodinis K. Computing optimal strategies for trees in linear time. In: Proceedings of the eighth annual European symposium on algorithms; 2000. pp. 403–4.