Optim Lett (2017) 11:1069–1089
DOI 10.1007/s11590-014-0793-z

ORIGINAL PAPER

General Variable Neighborhood Search for computing graph separators

Jesús Sánchez-Oro · Nenad Mladenović · Abraham Duarte

Received: 17 February 2014 / Accepted: 2 September 2014 / Published online: 18 September 2014
© Springer-Verlag Berlin Heidelberg 2014

Abstract Computing graph separators in networks has a wide range of real-world applications. For instance, in telecommunication networks, a separator determines the capacity and brittleness of the network. In the field of graph algorithms, the computation of balanced small-sized separators is very useful, especially for divide-and-conquer algorithms. In bioinformatics and computational biology, separators are required in grid graphs providing a simplified representation of proteins. This papers presents a new heuristic algorithm based on the Variable Neighborhood Search methodology for computing vertex separators. We compare our procedure with the state-of-the-art methods. Computational results show that our procedure obtains the optimum solution in all of the small and medium instances, and high-quality results in large instances.

Keywords Combinatorial optimization · Metaheuristics · VNS · Graph separators

1 Introduction

Let us consider an undirected connected graph $G = (V, E)$, where V represents the set of vertices ($|V| = n$) and E represents the set of edges ($|E| = m$). Let c_v be

J. Sánchez-Oro · A. Duarte (✉)
Dpto. de Ciencias de la Computación, Universidad Rey Juan Carlos, Móstoles, Spain
e-mail: abraham.duarte@urjc.es

J. Sánchez-Oro
e-mail: jesus.sanchezoro@urjc.es

N. Mladenović
LAMIH, Université de Valenciennes, Valenciennes, France
e-mail: nenad.mladenovic@brunel.ac.uk

the cost associated with each vertex $v \in V$ and let b be a positive integer such that $1 \leq b \leq n$. The Vertex Separator (VS) problem consists of dividing V into three non-empty subsets A,B, and C, such that there is not any edge between A and B, the size of both sets are bounded by b, and the sum of the weight of the vertices in C is minimized. In mathematical terms:

$$\begin{aligned} &\min \sum_{v \in C} c_v \\ &s.t. \\ &\qquad \max\{|A|, |B|\} \leq b \\ &\qquad A \cup B \cup C = V \\ &\qquad \forall u \in A \wedge \forall v \in B, (u, v) \notin E \end{aligned}$$

Notice that this formulation is equivalent to the maximization of the sum of the weight of the vertices in A and B (see for instance [21]). A solution $x = \{A, B, C\}$ verifies that $A \cup B \cup C = V$ and $A \cap B = A \cap C = B \cap C = \emptyset$. A separator C of G is then defined as a subset of vertices that divides G into two disconnected subgraphs. Graph separators have appeared under different names in the literature, including bifurcators, balanced cuts, or partitions.

Figure 1a shows an illustrative example of a graph G with 11 vertices and 14 edges. Figure 1b depicts a possible solution, where sets A, B, and C are represented as dashed ellipses, and the size constraint b is set to 4. If we assume, for the sake of simplicity, that each vertex $v \in V$ has an associated cost of $c_v = 1$, the value of this solution is 6 since there are six vertices in the separator (4, 6, 7, 8, 9, 11).

This optimization problem was originally introduced [2] in the context of Very Large Scale Integration (VLSI) design. However, finding balanced separators of small size have become an important task in several contexts. For instance, in telecommunication networks, a separator determines the capacity and brittleness of the network [10,12]. In the field of graph algorithms, the computation of balanced small-sized separators is very useful, especially for divide-and-conquer algorithms (see [14] for a larger description). In bioinformatics and computational biology, separators are required in grid graphs providing a simplified representation of proteins [8].

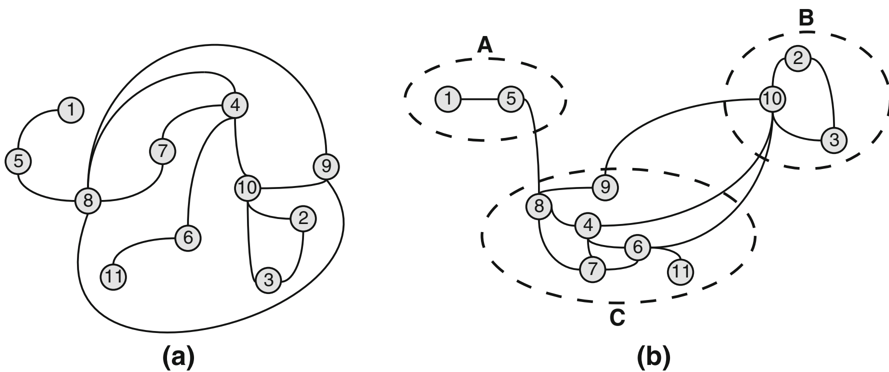

Fig. 1 **a** Example of a graph and **b** a possible VS solution

Finding the minimum vertex separator of a general graph is a NP-hard problem [4]. Therefore, exact approaches only solve instances of moderate size. In particular, de Souza and Balas [21] proposed a mixed integer programming formulation, and investigated the VS polytope and the convex hull of incidence vectors of vertex separators. This theoretical study was afterwards included in a branch-and-cut procedure. Computational results showed that the exact method was able to optimally solve instances with size ranging from 50 to 150 vertices in moderate computing time. Biha and Meurs [3] introduced new classes of valid inequalities and a simple lower bound. Computational experiments showed that the new exact procedure was able to solve to optimality all the instances introduced in de Souza and Balas [21], marginally reducing the computing time. Therefore, this method emerges as the state-of-the-art regarding exact procedures.

Much of the previous work on the VS problem is based on designing approximation algorithms. Lipton and Tarjan [13] showed that an n-vertex planar graph has a balanced separator of size $O(\sqrt{n})$. However, most of the papers approximate the VS problem by solving the Edge Separator (ES) problem. In particular, these approaches first design an approximation algorithm for the ES problem and then adapt this algorithm to the VS problem. See for instance, Leighton and Rao [11] or Arora et al. [1] with an approximation ratio of $O(\log n)$ and $O(\sqrt{\log n})$, respectively. This approach works well on graphs with bounded degree. However, for general graphs, the situation is completely different [6]. The best approximation algorithm for the VS problem was introduced in Feige et al. [7]. In particular, they proposed linear and semidefinite program relaxations, and rounding algorithms for these programs. They obtained an $O(\sqrt{\log(opt)})$ approximation, where opt is the size of the optimal separator.

Despite the fact that this problem has been the subject of extensive research, it has been mainly ignored in the heuristic and metaheuristic community. As far as we know, there is only one heuristic method introduced in de Souza and Balas [21]. In particular, the unique objective of this method is to generate feasible solutions for the branch-and-cut procedure. The method first solves the linear relaxation of the original problem. Then, starting from the fractional solution, it constructs an integer solution by sequentially rounding up (when it is possible) the fractional variables using a greedy function. After each rounding up, some variables are rounded down to 0 to fulfill the constraints of the problem.

In this paper, we propose a Variable Neighborhood Search (VNS) approach [22] for the Vertex Separator problem. This metaheuristic has been successfully applied to a large number of optimization problems. See for instance [5,15,19,20]. The rest of the paper is organized as follows: the Variable Neighborhood Search metaheuristic is described in Sect. 2, where we mainly focus on the General Variable Neighborhood Search variant. Sections 3 and 4 describe, respectively, the two constructive procedures designed for this problem as well as the three neighborhood structures. All the VNS strategies are experimentally tested in Sect. 5. Finally, the paper ends with the associated conclusions (Sect. 6).

2 Variable Neighborhood Search

Variable Neighborhood Search (VNS) is a metaheuristic for solving optimization problems based on systematic changes of neighborhood structures, without guaranteeing the optimality of the solution. 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 [9] for a complete review of this methodology and its different variants. In this paper, we focus on the General VNS variant (GVNS). See [17] for additional details. Algorithm 1 shows the pseudocode for GVNS.

Algorithm 1: GVNS(x, k_{max}, t_{max})

```
1: repeat
2:    k ← 1
3:    repeat
4:       x′ ← Shake(x, k)
5:       x″ ← VND(x′)
6:       NeighborhoodChange(x, x″, k)
7:    until k = k_max
8:    t ← CpuTime()
9: until t > t_max
```

The method has three input arguments: the initial solution (x), which can be a random solution or a solution given by a *constructive procedure*, the maximum neighborhood to be explored (k_{max}), and the maximum execution time allowed for the algorithm (t_{max}). The algorithm mainly consists of executing three methods: *shake*, *VND*, and *neighborhood change*. Firstly, given a solution x, the shake method generates a new solution, x', in the k neighborhood of the current solution (step 4). Then, it is is improved using a VND method, producing a new improved solution x'' (step 5). Finally, the neighborhood change method decides whether the new solution x'' improves upon x or not (step 6). If so, the search returns to the first neighborhood ($k = 1$) and updates the incumbent solution ($x = x''$); otherwise, the VNS metaheuristic tries to find a better solution in a different neighborhood ($k = k + 1$). We describe in the following sections the strategies designed for solving the VS problem based on GVNS approach.

In this paper we propose two constructive procedures (C1 and C2) for the VS problem (see Sect. 3). Specifically, C1 favors the diversification of the search (it is mainly a random procedure) while C2 focuses on the intensification of the search (by means of an elaborated heuristic approach). We additionally investigate three different neighborhoods for this problem, as well as two local searches and two VND methods derived from them (see Sect. 4).

We propose a standard shake procedure based on a random perturbation of the incumbent solution since its main objective is to diversify the search. However, a straightforward implementation of a shake procedure, $Shake(x, k)$ could produce infeasible solutions. In particular, all moves that consist of changing one vertex from A or B to C always produces a (worse) feasible solution. On the other hand, a move

from one vertex in C to A or B could produce (a better) infeasible solution. Therefore, we propose to use in our shake procedure only moves from A or B to C since it is expected that the improvement strategy produces a better solution in subsequent steps of the VNS.

The function *neighborhood change* basically compares the new value of x' with the incumbent value of x obtained in the neighborhood k. If an improvement is obtained, k is reset to its original value (usually $k = 1$). Otherwise, the next neighborhood is considered for a further exploration (usually $k = k + 1$). In this paper, we investigate the effect on the search of the so-called jump neighborhood search. In this case, the corresponding function considers two additional parameters, k_{min} and k_{step}, that control the change of the neighborhood. Specifically, when the GVNS method performs an improving move it sets $k = k_{min}$ instead of $k = 1$. Similarly, in non-improving moves it sets $k = k + k_{step}$ instead of $k = k + 1$. We will investigate the effect of these two parameters in Sect. 5.

3 Constructive procedures

VNS is a trajectory-based metaheuristic which mainly focuses on the improving of an available solution. Most of the VNS implementations consider a random solution as the starting point of the search. In this paper, we will investigate the effect of constructing a solution at random or using a more elaborated heuristic procedure. The first constructive method, C1, basically constructs, as fast as possible, a feasible solution. In particular, C1 starts by initializing $A = B = C = \emptyset$. Then, a vertex is randomly selected from V and inserted, if it is possible, in either A or B. Otherwise, the vertex in inserted in C. This process is repeated until all vertices in V are explored, returning a feasible random solution.

The other procedure, C2, receives as input parameter a tree-level structure [5], which is a partition of a graph in different sets $L_1, L_2, \ldots, L_t$, called levels. The first level contains only one vertex, which will be the root of the level structure. The rest of levels L_l contain all the vertices adjacent to any vertex in the previous level, L_{l-1}, that are not present in any L_j, with $1 \leq j < l - 1$. The number of levels will vary depending of the graph and the vertex selected as root of the level structure. This level structure guarantees that vertices in alternative levels are not adjacent. If we would ignore the size constraint (i.e., $|A| \leq b$ and $|B| \leq b$) each level L_l would be a separator ($C = L_l$) of the graph, where A would contain the vertices in levels i, with $1 \leq i < l$, B would contain vertices in levels j, with $l < j \leq t$, and finally C would contain the vertices in the selected level l. When considering the size constraint, obviously not all levels would become a separator by itself since the number of vertices in either A or B may exceed b, producing an infeasible solution. In order to overcome this situation, C2 selects as the separator the level which divides the vertices of the graph into two sets of roughly equal size. Algorithm 2 shows the pseudocode of this constructive procedure.

The method starts by including all the levels of the tree in the separator (step 1). Additionally, two scanning variables (i and j) are set to the first and last levels respectively (see steps 2 and 3). These two variables allow the method to traverse the

tree in top-down (i) or bottom-up (j) directions. We also include a boolean variable ($finished$) which verifies if the solution under construction is finished or not.

The algorithm performs iterations (steps 5–17) until the top-down index reaches the bottom-up index plus one (since we need to reserve at least one level for the separator) or the boolean variable is set to `true`. The while-loop basically inserts one complete level in A (symmetrically B) if its size is smaller than B (symmetrically A). In addition, it is checked whether the inclusion of the corresponding complete level (in A or B) produces a feasible solution or not. If not, the boolean variable is set to `true` (step 15) and the method finishes returning the constructed solution.

Algorithm 2: C2$(L_1, \ldots, L_t)$

1: $\{A, B, C\} \leftarrow \left\{\emptyset, \emptyset, \bigcup_{i=1}^{t} L_i\right\}$
2: $i \leftarrow 1$
3: $j \leftarrow t$
4: *finished* $\leftarrow$ FALSE
5: **while** $(i < j)$ **and not** *finished* **do**
6:    **if** $(|A| \leq |B|)$ **and** $(A \cup L_i \leq b)$ **then**
7:       $A \leftarrow A \cup L_i$
8:       $C \leftarrow C \setminus L_i$
9:       $i \leftarrow i + 1$
10:    **else if** $(|B| < |A|)$ **and** $(B \cup L_j \leq b)$ **then**
11:       $B \leftarrow B \cup L_j$
12:       $C \leftarrow C \setminus L_j$
13:       $j \leftarrow j - 1$
14:    **else**
15:       *finished* $\leftarrow$ TRUE
16:    **end if**
17: **end while**
18: **return** $\{A, B, C\}$

Considering the graph presented in Fig. 1a (with $|V| = 11$, $|E| = 19$, and $b = 4$), a possible tree-level is shown in Fig. 2a, where vertex 1 is selected as the root of the tree. Notice that each vertex in a given level is adjacent to other vertices in the same level or in consecutive levels (but never in alternative levels). Again, c_v has been set to 1 for all vertices.

Figure 2b shows the the first step of the algorithm, where all the levels are initially placed at the separator while sets A and B remain empty. C2 starts by including the first level (root) in set A (Fig. 2c) and then the last level is inserted in set B (Fig. 2d). Considering that $|B| > |A|$ we continue inserting levels in A (Fig. 2e, f). In this situation, we have two levels in C that cannot be inserted in either A or in B (since the addition of a level would violate the size constraint). Therefore, the constructive procedure finishes by returning the associated solution. Finally, if the last level contains more than b vertices, the solution would be infeasible. Then, C2 assigns as much vertices as possible to B, and the remaining ones are assigned to C.

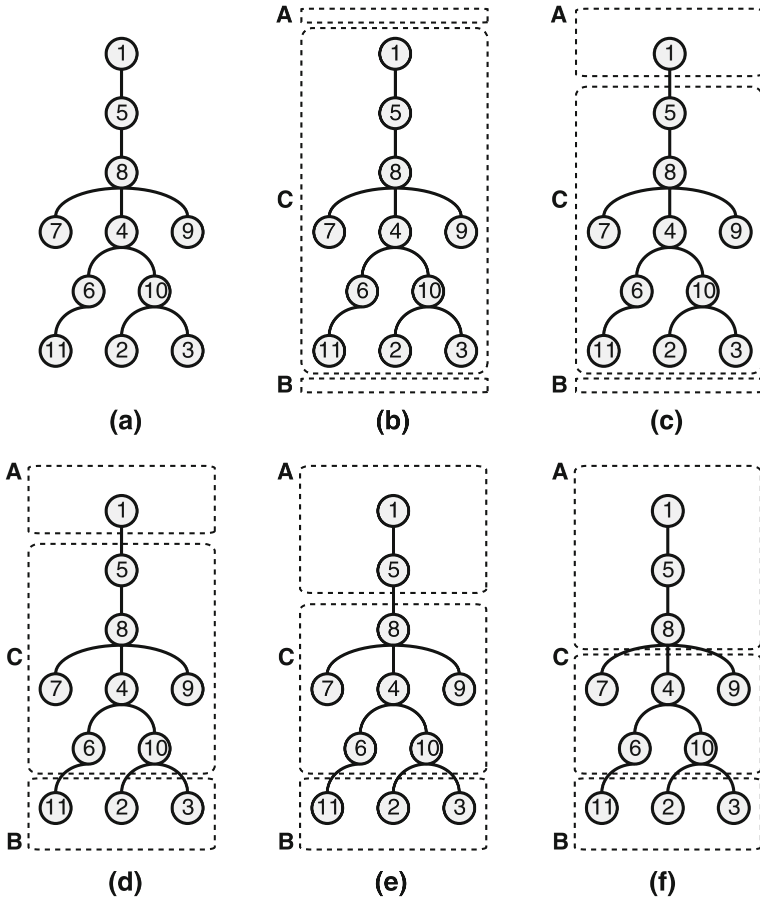

Fig. 2 Example of a solution constructed with the greedy procedure C2

4 Local search methods

Solutions to the VS problem are typically represented as three different disjoint sets (i.e., $x = \{A, B, C\}$). In this section we define three different neighborhood structures based on three different moves. In particular, the neighborhood N_1 contains the neighbor solutions obtained by removing one vertex v from A or B and inserting it in C. We denote this move as $Move_1$ and it is formally described as follows:

$$Move_1(v, \{A, B, C\}) = \begin{cases} A \leftarrow A \backslash \{v\} \\ C \leftarrow C \cup \{v\} \end{cases} \text{if } v \in A \quad \begin{cases} B \leftarrow B \backslash \{v\} \\ C \leftarrow C \cup \{v\} \end{cases} \text{if } v \in B$$

Figure 3 illustrates this type of move. In particular, Fig. 3a shows the original graph where the vertex involved in the move is highlighted (vertex 8). Figure 3b shows the

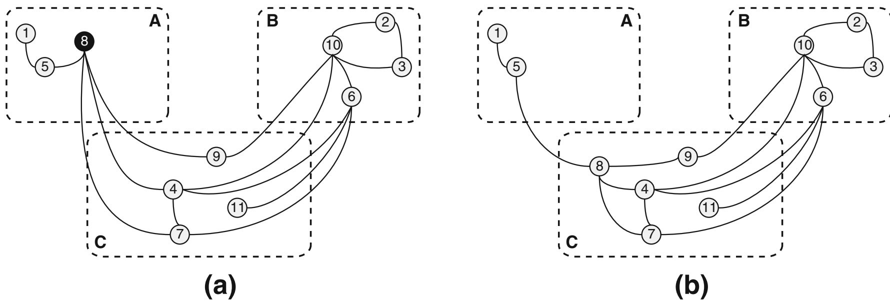

Fig. 3 Example of the $Move_1$. **a** Original solution, **b** resulting solution

resulting solution. It is easy to see that this move always produces feasible solutions (except the case of $|A| = 1$, since it is mandatory to have at least one element in A).

The second neighborhood N_2 contains all solutions produced by removing one vertex from A (symmetrically from B) and inserting it in B (symmetrically in A). In this case, we could obtain an infeasible solution. In order to overcome this situation, if we move a vertex $v \in A$ to B we additionally move the adjacent vertices (in the corresponding set) to the separator. This move, denoted as $Move_2$, is mathematically described as follows:

$$Move_2(v, \{A, B, C\}) = \begin{cases} A \leftarrow A \backslash \{v\} \backslash N_A(v) & \\ B \leftarrow B \cup \{v\} & \text{if } v \in A \\ C \leftarrow C \cup N_A(v) & \\ B \leftarrow B \backslash \{v\} \backslash N_B(v) & \\ A \leftarrow A \cup \{v\} & \text{if } v \in B \\ C \leftarrow C \cup N_B(v) & \end{cases}$$

where $N_A(v)$ (simmetrically $N_B(x)$) contains the adjacent vertices of v contained in A (i.e., $N_A(v) = \{u \in A : (u, v) \in E\}$). Figure 4 illustrates this type of move. Specifically, Fig. 4a shows the original graph where the vertex involved in the move is highlighted (vertex 8). Figure 4b shows the resulting solution which is infeasible, since there exists an edge between A and B (edge $(5, 8)$). In order to make this solution feasible, the adjacents of vertex 8 in A (i.e., $N_A(8) = \{5\}$) must be moved from A to C. After that, the resulting solution is feasible (see Fig. 4c).

The third neighborhood N_3 contains the set of solutions produced by removing one vertex from the separator C and inserting it in either A or B. This move is denoted as $Move_3$. In order to maintain the feasibility of the solutions, we follow a similar strategy than the one in N_2. Therefore, we formally define this move as follows:

$$Move_3(v, \{A, B, C\}) = \begin{cases} A \leftarrow A \cup \{v\} & \\ B \leftarrow B \backslash N_B(v) & \text{if } v \text{ is inserted in } A \\ C \leftarrow C \backslash \{v\} \cup N_B(v) & \\ A \leftarrow A \backslash N_A(v) & \\ B \leftarrow B \cup \{v\} & \text{if } v \text{ is inserted in } B \\ C \leftarrow C \backslash \{v\} \cup N_A(v) & \end{cases}$$

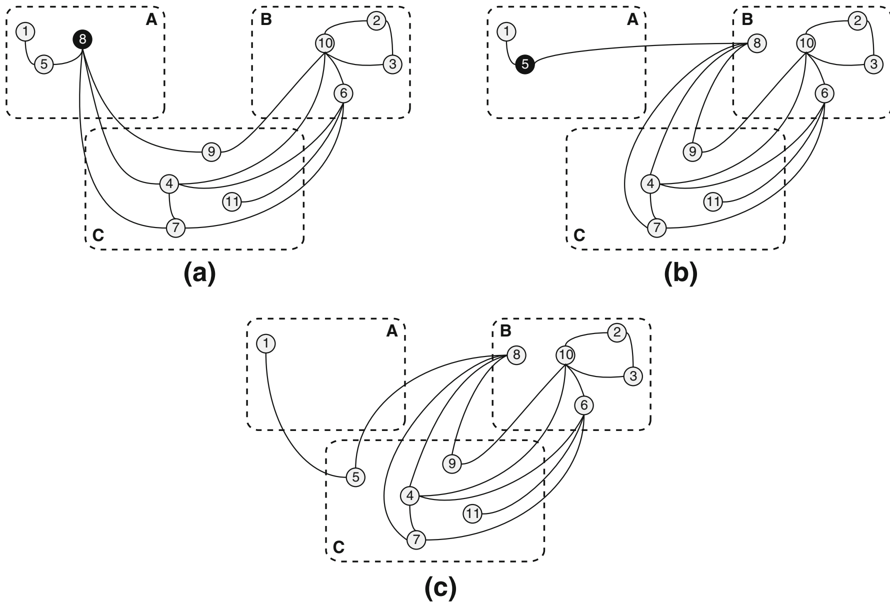

Fig. 4 Example of the $Move_2$. **a** Original solution, **b** intermediate solution, **c** resulting solution

Figure 5 shows an example of this type of move. The first figure highlights the involved vertex in the move (vertex 9). Figure 5b shows the resulting intermediate infeasible solution after removing vertex 9 from C and inserting it in A. In order to ensure the feasibility of all solutions in N_3 [see edge (9, 10) in Fig. 5b], all the vertices adjacent to 9 in set B (i.e., $N_B(9) = \{10\}$) must be included in the separator (Fig. 5c).

The effectiveness of a move directly depends on how it changes the value of the objective function. In particular, the VS problem is a minimization problem so any move which reduces the number of vertices in the separator is an improving move. Given a solution $\{A, B, C\}$, we can predict the change of the objective function if we move vertex v with any of the three different moves defined above. In order to simplify the $MoveValue$ description, the vertex weights are assumed to be 1. This prediction is denoted as $MoveValue_i(v, \{A, B, C\})$ where the value of i indicates the type of move performed (with $1 \leq i \leq 3$). Considering $Move_1$, the associated move value is 1 since one vertex is removed from either A or B and inserted in C. Therefore, it always deteriorates the value of the objective function in one unit ($MoveValue_1(v, \{A, B, C\}) = 1, \forall v \in V$).

$Move_2$ considers moves of vertices between A and B. Then, the variation produced in the objective function is:

$$MoveValue_2(v, \{A, B, C\}) = \begin{cases} |N_A(v)| \text{ if } v \in A \\ |N_B(v)| \text{ if } v \in B \end{cases}$$

This value depends on $|N_A(v)|$ or $|N_B(v)|$, which are always larger than or equal to zero. Therefore, the new solution can only be equal or worse than the original one.

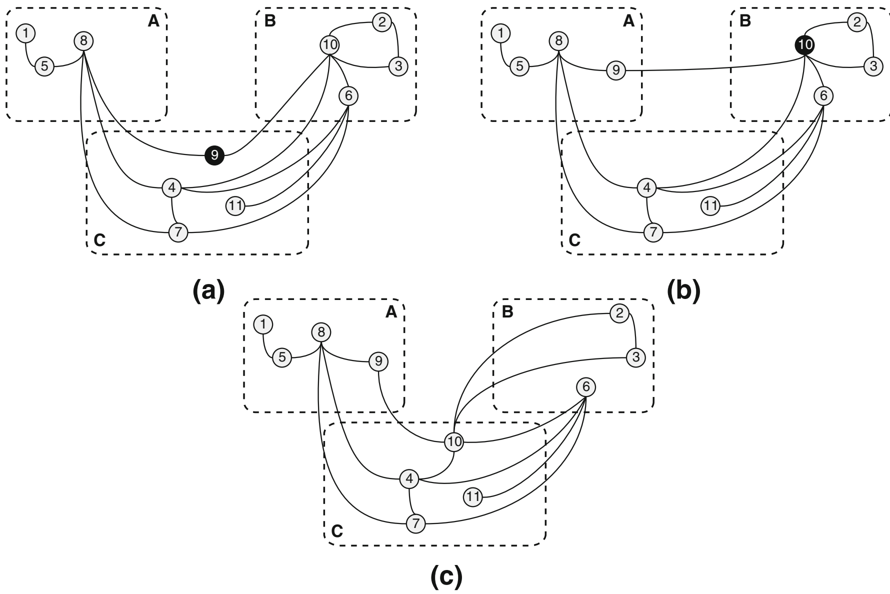

Fig. 5 Example of the $Move_3$. **a** Original solution, **b** intermediate solution, **c** resulting solution

Finally, $Move_3$ considers moves between the separator and A or B. In this situation, the associated move value is defined as:

$$MoveValue_3(v, \{A, B, C\}) = \begin{cases} |N_B(v)| - 1 \text{ if } v \in A \\ |N_A(v)| - 1 \text{ if } v \in B \end{cases}$$

The quality of the new solution again depends on the value of $|N_A(v)|$ or $|N_B(v)|$. Therefore, the best possible move is able to improve the current solution by only one unit (this implies that $|N_A(v)| = 0$ or $|N_B(v)| = 0$).

Taking into account the definition of move values given above, we will not be able to find better solutions by exploring N_1 or N_2 in isolation. For that reason, we proposed two local search strategies LS_1 and LS_2 based on a nested exploration strategy of composed neighborhoods. In particular, LS_1 considers N_1 and N_3, while LS_2 considers N_2 and N_3. Algorithm 3 shows the pseudo-code of LS_1. We do not include the pseudocode of LS_2 since it is similar when $Move_1$ is replaced by $Move_2$ (step 6). The algorithm first identifies the set of candidate vertices, set S, to be moved in N_1 (step 3). Then, the corresponding neighborhood is explored (steps 4–16), where the vertex to be moved is selected at random (step 5). Notice that we have tested other strategies to select vertices with insignificant influence in the outcomes. The move is performed in step 6, producing a new solution $\{A', B', C'\}$ in N_1. After that, the local search intensively explores N_3 (steps 7–12), updating the best solution found (step 9). If we find a better solution in N_3, the incumbent solution is updated (step 14) and the algorithm performs a new iteration. Otherwise, the local search tries to move the next vertex in set S. The algorithm ends when no improvement is found.

Algorithm 3: LS1($\{A, B, C\}$)

```
1: improved ← FALSE
2: while not improved do
3:   S ← A ∪ B
4:   while not improved and S ≠ ∅ do
5:     v ← Select(S)
6:     {A', B', C'} ← Move_1(v, {A, B, C})
7:     for all u ∈ C' do
8:       if MoveValue_3(u, {A', B', C'}) < 0 then
9:         {A', B', C'} ← Move_3(u, {A', B', C'})
10:        improved ← TRUE
11:      end if
12:    end for
13:    if improved then
14:      {A, B, C} ← {A', B', C'}
15:    end if
16:  end while
17: end while
```

We further investigate the exploration of the three neighborhoods by using a Variable Neighborhood Descent (VND) procedure. As it was described above, given a solution, it is not possible to find a better one in neither N_1 nor N_2. Therefore, the VND uses the composed neighborhoods described above. VND obtains a local optimum with respect to the first composed neighborhood. Then, instead of abandoning the search (as a local search procedure), VND resorts to the other composed neighborhood searching for an improvement. If we find an improving move, we again start exploring the first composed neighborhood. At the end of the search VND ensures that the returned solution is a local optimum with respect to both composed neighborhoods.

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.

5 Computational experiments

This section reports the computational experiments that we performed for testing the effectiveness and efficiency of the proposed VNS for solving the VS problem. The algorithm was implemented in Java SE 7 and the experimentation was conducted on an Intel Core i7 2600 CPU (3.4 GHz) and 4 GB RAM. The proposed algorithm was tested over the current VS benchmark which consists on 104 instances (with size ranging from 11 to 191) with known optimum. These instances are adapted [21] from the well-known Matrix Market benchmark,[1] in order to produce difficult instances for the VS problem. De Souza and Balas [21] confirm that the hardness of the instances increases with the number of vertices and density. Additionally, a relevant parameter that also affects to the difficulty is the value of the size constraint b. Specifically, they

[1] http://math.nist.gov/MatrixMarket/.

suggest to set this factor to $2n/3$. A detailed description of these instances can be found in de Souza and Balas [21].

The experiments are divided into two stages. In the first stage we use a subset of 25 representative instances of the current VS benchmark to test our algorithms and tune their parameters (k_{step} and k_{max}). In the second stage, once our best algorithm is identified, we compare it with the best previous methods, proposed in [3,21]. Both previous algorithms are exact methods, so we test the time needed by our approach to obtain the optimum value and compared with the time needed by the exact methods. Tables 1, 2, 3, 4, 5 and 6 report the average quality over all instances, Avg.; the CPU time in seconds, Time (s); the average percent deviation with respect to the optimum value, Dev. (%); and the number of optimum values found, #Opt.

In the first experiment we compare the performance of the proposed constructive procedures (C1 and C2) described in Sect. 3. We generate one solution for each instance of the preliminary testbed with each constructive procedure. The root for tree-based structure for C2 is selected at random. Table 1 shows that C1 is able to obtain a lower deviation than C2. However, C2 obtains one more optimum solution than C1. The time needed by both methods to construct a solution is neglectable (less than 0.01 s). We then select both constructive methods to analyze their performance coupled with the local search methods, since it is not possible to identify the best algorithm.

In the next experiment we test the performance of the two constructive procedures coupled with the local search methods proposed, LS1 and LS2 (see Sect. 4). For each instance, we have constructed one solution and improved it with the corresponding local search. Table 2 shows that C2 coupled with LS1 obtains the best average quality, as well as largest number optimum solutions found. The time needed to improve a solu-

Table 1 Comparison of the proposed constructive procedures

	Avg.	Time (s)	Dev. (%)	#Opt
C1	26.88	0.00	72.02	8
C2	31.12	0.00	92.01	9

Table 2 Comparison of the constructive procedures coupled with the local search methods

	Avg.	Time (s)	Dev. (%)	#Opt
C1+LS1	23.08	0.01	22.58	11
C1+LS2	22.60	0.01	14.90	13
C2+LS1	22.44	0.00	21.66	16
C2+LS2	22.52	0.00	12.36	13

Table 3 Performance of the best local search method (LS1) with different number of iterations

	Avg.	Time (s)	Dev. (%)	#Opt
C2+LS1 (1)	22.44	0.01	21.66	16
C2+LS1 (25)	20.80	0.07	1.08	21
C2+LS1 (50)	20.76	0.11	0.41	22
C2+LS1 (100)	20.72	0.24	0.30	23

Table 4 Performance of the two VND algorithms proposed and the best local search

	Avg.	Time (s)	Dev. (%)	#Best
C2+LS1	20.72	0.24	0.30	23
C2+VND1	20.68	0.53	0.12	24
C2+VND2	20.68	0.55	0.20	24

Table 5 Performance of GVNS using different values of k_{step} and k_{max}

k_{max}	k_{step}	Avg.	Time (s)	Dev. (%)	#Best
0.15	0.01	20.68	4.53	0.12	24
	0.02	20.68	4.50	0.12	24
	0.05	20.68	4.49	0.12	24
0.25	0.01	20.72	4.54	0.24	24
	0.02	20.68	4.52	0.12	24
	0.05	20.68	4.51	0.12	24
0.50	0.01	20.68	4.65	0.12	24
	0.02	20.64	4.56	0.00	25
	0.05	20.68	4.52	0.12	24

Table 6 Performance of GVNS compared with the best previous algorithms

	Avg.	Time (s)	Dev. (%)	#Opt
GVNS	20.89	4.78	0.00	104
Souza and Balas [21]	20.92	213.52	0.08	103
Biha and Meurs [3]	20.89	171.22	0.00	104
CPLEX	20.90	9.35	0.03	103

tion is again neglectable. We identify C2+LS1 as the best combination of constructive plus local search.

The third experiment studies the influence of performing several independent executions instead of constructing only one solution. This strategy is usually denoted as Multi-Start methods (see [16]). The constructive procedures proposed in this paper are not deterministic, so each construction will find a different solution. Specifically, C2 is able to obtain $|V|$ different solutions, depending on the vertex used as the root of the level tree. Table 3 shows the performance of C2+LS1 when considering 1, 25, 50, and 100 independent iterations. As expected, the CPU time increases with the number of iterations. However, this experiment shows that the computing time linearly grows with the number of iterations. What it is even more important, the deviation and the number of optima are significantly improved (without expending large running times). Then, we consider a multi-start strategy with 100 independent iterations for the rest of the experimentation.

In the fourth preliminary experiment, we compare the performance of the VND with the best local search in isolation (LS1). We additionally compare the order of exploration of the corresponding neighborhoods. In particular VND1 firstly explores

the composed neighborhood N_1–N_2 and then N_2–N_3, while VND2 performs the symmetric exploration. Results in Table 4 confirm that both VND procedures compare favorably with the simple local search method. Additionally, considering the results of each VND variant, we conclude that VND2 emerges as the best procedure since it presents the best deviation with similar CPU time.

The next experiment consists of selecting the best GVNS variant by running a single full-factorial experiment. In particular, we consider $k_{step} = \{0.01n, 0.02n, 0.05n\}$ and $k_{max} = \{0.15n, 0.25n, 0.50n\}$ where n identifies the number of vertices of the instance. Notice that each GVNS variant considers C2, VND2 and a multi-start strategy as concluded above. Results in Table 5 show that the difference among the variants are relatively small. We select $k_{step} = 0.02n$ and $k_{max} = 0.50n$ since the associated GVNS variant reaches all the optima in the considered set of instances. We therefore select this procedure to be used in further comparisons

In the second part of this section, we compare our best GVNS procedure with the current state of the art in the VS problem. In particular, with the exact methods presented in [3,21], using the set of 104 instances reported in these papers. The results obtained by these two methods have been directly taken from the original source (where maximum computing time was set to 1,800 s). We have also included the in this Table the results obtained by CPLEX 12 (executing the mathematical model introduced in [21]), using the same time limit. If an exact method does not finish the exploration within the time limit, it returns the best solution found without guaranteeing the optimality.

Table 6 shows the results obtained in this experiment. As we can see, our method is able to find all optima in less than 5 s (notice that the GVNS does not guarantee the optimality of the solution). On the other hand, de Souza and Balas does not find the optimal solution in one instance. Additionally, it does not guarantee the optimality in five more instances. It is important to remark that the method is almost 50 times slower than our GVNS. The algorithm of Biha and Meurs presents better performance since it is able to find all the optima in the considered instances. However, the computing time is considerably larger (35 times) than time needed by th GVNS. CPLEX is able to optimally solve 100 out of 104 instances. Analyzing the remaining four instances, CPLEX finds the optimal solution in three of them, but it is not able to certify it. It is important to remark that in these four instances CPLEX finishes its execution since the exploration tree does not fit in memory. Finally, notice that the computing time of CPLEX is really competitive, requiring only 9 s on average to reach the optimal value.

We would like to highlight that is is really difficult to compare running times of algorithm executed in different computers, implemented by different programmers and even different programming language. Therefore, the time analysis must be considered as a qualitative comparison.

In sight of the results obtained in the previous experiment, we can conclude that this set of instances is easily solved by our method and even by CPLEX. 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. We therefore propose to use two new set of instances well-studied in the optimization literature:

- Barabasi–Albert (BA): this set of is instances is constructed with an algorithm for generating random scale-free networks using a preferential attachment mechanism.

These networkds are widely observed in natural and human-made systems, including the Internet, the world wide web, citation networks, and some social networks. We construct 95 instances where the number of vertices is $100 \leq n \leq 1,000$ and the node degree is randomly selected from $[1, n]$.
- Erdos–Renyi (ER): this set contains instances (graphs) where the corresponding vertices are randomly connected. Each graph is usually denoted as $G(n, p)$, where n indicates the number of vertices and p represents the probability of connecting a pair of vertices (independent from every other edge). Obviously, the larger the value of p, the denser the associated graph. We construct 95 instances where the number of vertices ranges from 100 to 1,000 and the probability is randomly selected from [0.2, 1.0] (ensuring that the resulting graph is connected).

We refer the reader to http://www.optsicom.es/vs where these instances are publicly available as well as other relevant information (best known solution, execution time, etc.). As it is recommended by [21], the maximum size of the separator is set to $b = 2n/3$. These instance are actually a real challenge for modern heuristic methods.

In the last experiment we compare our best GVNS variant with CPLEX (executed again for a maximum computing time of 1,800 s). We do not provide the results of the two branch-and-bound procedures, since we do not have the executable code. However, as it is shown in the previous experiment, the results of CPLEX and the two branch-and-bound procedure are comparable. Therefore, it is expected that both exact procedure would have a performance similar to CPLEX.

Table 7 reports a summary of the results obtained by both procedures. Tables 8 and 9 in the Appendix present the individual results for each instance of Barabasi–Albert and Erdos–Renyi sets, respectively. We consider an additional column (#Best) to indicate the number of times that either GVNS or CPLEX match the best solution since the optimal value is not known for the majority of these instances. Notice that CPLEX does not guarantee the optimality when (1) the maximum computing time is exceeded or (2) the exploration tree does not fit in memory.

Attending to these results, we can concluded that GVNS consistently produces better results than CPLEX in both sets. In particular, in Barabasi–Albert set, GVNS reaches the same number of optimal solutions (10 out of 95) than CPLEX. In the remaining 85 instances, our procedure only misses 13 best solutions (obtaining the best solution in 72 out of 85 instances). The average deviation (0.16 %) and average objective function (279.25) also compares favorably with those obtained by CPLEX (1.15 % and 283.16). Finally, GVNS is almost 30 faster. The results in the Erdos–Renyi confirm the superiority of our method. Specifically, when the optimum is known (5

Table 7 Comparison of GVNS with CPLEX 12 in all set of instances

	Algorithm	Avg.	Time (s)	Dev. (%)	#Best	#Opt
Barabasi-Albert	CPLEX	283.16	1,064.13	1.15	46	10
	GVNS	279.25	36.56	0.16	72	10
Erdos-Renyi	CPLEX	296.57	885.52	1.23	53	5
	GVNS	292.02	33.89	0.10	86	5

instances), GVNS finds it. In the remaining instances, GVNS obtains the best result in 86 (out of 90) instances, achieving lower average deviation (0.16 vs. 1.15 %) and average objective function (292.02 vs. 296.57). Again the computing time of the propose procedure is considerably shorter (33.89 vs. 885.52 s).

6 Conclusions

In this paper we have proposed a General Variable Neighborhood Search (GVNS) for the Vertex Separator (VS) problem. As far as we know, it is the first heuristic procedure for general graphs. Specifically, we have introduced two constructive procedures, the first one basically constructs a feasible random solution, while the other constructs a tree-level structure. We also propose three different neighborhoods based on feasible moves. Considering the structure of a VS solution, we conclude that the exploration of those neighborhood in isolation does not drive to high quality solutions. Therefore, we present an exploration of the neighborhood in pairs using two different local search algorithms. We also investigate a deeper exploration of those composed neighborhoods by means of using a Variable Neighborhood Descendant (VND) approach. We embed the most promising strategies in GVNS, when we study the effect of using a jump neighborhood change function. The proposed methods have been tested on a large benchmark consisting of 104 well-known instances in the existing literature. Specifically, the GVNS has been able to find the optima in all of them. We finally propose two new sets of harder and challenging 190 instances to be the new benchmark for future comparisons of modern heuristics in solving the VS problem.

Acknowledgements This research has been partially supported by the Spanish Ministry of "Economía y Competitividad", Grant Ref. TIN2012-35632-C02-02.

7 Appendix

Tables 8 and 9 report the individual result obtained by the GVNS method. Each column shows, respectively, the vertex separator value (VS), computing time (Time) in seconds, number of vertices (n), number of edges (m), value of b and the construction parameter of the instance: node degree parameter used in the construction of the instance (N. deg.) for the Barabasi–Albert instances and the link probability parameter used in the construction of the instance (prob.) for the Erdos–Renyi instances. The maximum computing time in both sets is limited to $0.05 \cdot n$ s. The optimum values are highlighted in bold font.

Table 8 Results on Barabasi–Albert instances

Instance	VS	Time (s)	n	m	b	N. deg.	Instance	VS	Time (s)	n	m	b	N. deg.
B_A_1(100,65)	**43**	**5.13**	**100**	**3,619**	**66**	**65**							
B_A_1(1000,878)	564	93.57	1,000	454,234	666	878	B_A_3(550,294)	217	35.87	550	96,413	366	294
B_A_1(150,137)	**86**	**7.64**	**150**	**10,458**	**100**	**137**	B_A_3(600,435)	293	35.01	600	142,652	400	435
B_A_1(200,175)	112	10.11	200	18, 030	133	175	B_A_3(650,642)	387	41.02	650	209,008	433	642
B_A_1(250,146)	99	13.01	250	21,173	166	146	B_A_3(700,678)	418	37.65	700	238,905	466	678
B_A_1(300,255)	160	16.01	300	39,899	200	255	B_A_3(750,643)	416	38.40	750	251,152	500	643
B_A_1(350,320)	198	17.98	350	57,220	233	320	B_A_3(800,595)	409	59.20	800	258,185	533	595
B_A_1(400,376)	234	20.86	400	76,323	266	376	B_A_3(850,693)	458	65.65	850	311,299	566	693
B_A_1(450,326)	218	23.45	450	80,095	300	326	B_A_3(900,851)	535	54.22	900	388,341	600	851
B_A_1(500,277)	204	25.16	500	81,802	333	277	B_A_3(950,553)	401	62.09	950	305,620	633	553
B_A_1(550,499)	314	33.41	550	140,558	366	499	**B_A_4(100,87)**	**51**	**5.10**	**100**	**4,452**	**66**	**87**
B_A_1(600,541)	349	32.74	600	166,631	400	541	B_A_4(1000,509)	391	78.21	1,000	307,697	666	509
B_A_1(650,465)	320	45.84	650	165,831	433	465	B_A_4(150,111)	71	7.55	150	8,953	100	111
B_A_1(700,649)	415	40.76	700	231,521	466	649	**B_A_4(200,197)**	**127**	**10.11**	**200**	**19,699**	**133**	**197**
B_A_1(750,422)	303	59.74	750	186,121	500	422	B_A_4(250,133)	98	13.23	250	19,882	166	133
B_A_1(800,627)	418	59.07	800	267,812	533	627	B_A_4(300,205)	139	17.08	300	34,135	200	205
B_A_1(850,619)	418	76.26	850	287,289	566	619	B_A_4(350,294)	188	17.53	350	53,828	233	294
B_A_1(900,817)	527	59.72	900	376,975	600	817	B_A_4(400,350)	225	22.92	400	72,456	266	350
B_A_1(950,626)	444	57.83	950	334,026	633	626	B_A_4(450,229)	165	26.05	450	62,237	300	229
B_A_2(100,69)	**45**	**5.05**	**100**	**3,791**	**66**	**69**	B_A_4(500,496)	305	26.78	500	124,031	333	496
B_A_2(1000,856)	556	100.28	1,000	446, 192	666	856	B_A_4(550,347)	245	31.13	550	108,420	366	347
B_A_2(150,94)	65	7.66	150	8, 057	100	94	B_A_4(600,305)	226	31.05	600	110,549	400	305
B_A_2(200,161)	105	10.17	200	17,034	133	161	B_A_4(650,535)	347	36.09	650	183,074	433	535
B_A_2(250,235)	**147**	**12.50**	**250**	**29,738**	**166**	**235**	B_A_4(700,621)	395	45.10	700	224,315	466	621

Table 8 continued

Instance	VS	Time (s)	n	m	b	N. deg.	Instance	VS	Time (s)	n	m	b	N. deg.
B_A_2(300,220)	148	15.11	300	35,846	200	220	B_A_4(750,722)	453	42.79	750	273,170	500	722
B_A_2(350,182)	129	21.29	350	38,250	233	182	B_A_4(800,750)	477	59.90	800	304,973	533	750
B_A_2(400,227)	165	23.78	400	53,219	266	227	B_A_4(850,646)	434	72.54	850	296,126	566	646
B_A_2(450,392)	252	25.52	450	91,412	300	392	B_A_4(900,768)	510	68.77	900	360,454	600	768
B_A_2(500,288)	205	36.97	500	84,122	333	288	B_A_4(950,758)	507	89.51	950	382,879	633	758
B_A_2(550,355)	247	31.19	550	110,444	366	355	**B_A_5(100,89)**	**55**	**5.05**	**100**	**4547**	**66**	**89**
B_A_2(600,520)	335	35.86	600	161,917	400	520	B_A_5(1000,578)	413	74.64	1, 000	336,964	666	578
B_A_2(650,485)	327	41.45	650	170,756	433	485	B_A_5(150,103)	67	7.64	150	8,513	100	103
B_A_2(700,545)	368	47.10	700	204,380	466	545	**B_A_5(200,199)**	**132**	**10.15**	**200**	**19,833**	**133**	**199**
B_A_2(750,395)	293	61.85	750	177,319	500	395	B_A_5(250,132)	94	13.07	250	19,726	166	132
B_A_2(800,617)	419	54.16	800	265,081	533	617	B_A_5(300,211)	139	16.11	300	34,781	200	211
B_A_2(850,739)	478	53.72	850	325,755	566	739	B_A_5(350,249)	168	18.01	350	47,720	233	249
B_A_2(900,576)	411	49.20	900	293,353	600	576	B_A_5(400,233)	162	23.82	400	54,130	266	233
B_A_2(950,744)	505	91.86	950	377,886	633	744	B_A_5(450,424)	270	25.17	450	96,757	300	424
B_A_3(100,64)	**43**	**5.10**	**100**	**3,561**	**66**	**64**	B_A_5(500,408)	270	27.48	500	107,691	333	408
B_A_3(1000,601)	430	72.14	1,000	346,978	666	601	B_A_5(550,495)	317	31.76	550	139,977	366	495
B_A_3(150,129)	**83**	**7.62**	**150**	**10,038**	**100**	**129**	B_A_5(600,475)	316	30.97	600	151,797	400	475
B_A_3(200,111)	81	10.66	200	13,099	133	111	B_A_5(650,434)	304	47.87	650	157,753	433	434
B_A_3(250,191)	124	13.26	250	25,633	166	191	B_A_5(700,501)	346	35.70	700	192,274	466	501
B_A_3(300,260)	159	15.56	300	40,405	200	260	B_A_5(750,744)	453	43.28	750	279,231	500	744
B_A_3(350,251)	166	17.98	350	47,999	233	251	B_A_5(800,663)	432	49.66	800	278,900	533	663
B_A_3(400,284)	193	22.92	400	62,401	266	284	B_A_5(850,635)	433	71.70	850	292,512	566	635
B_A_3(450,243)	179	24.86	450	65,013	300	243	B_A_5(900,662)	452	88.55	900	324,198	600	662
B_A_3(500,273)	200	25.12	500	80,883	333	273	B_A_5(950,818)	534	83.49	950	404,427	633	818

Table 9 Results on Erdos–Renyi instances

Instance	VS	Time (s)	n	m	b	prob.	Instance	VS	Time (s)	n	m	b	prob.
E_R_1(100,0.89)	**82**	**5.00**	**100**	**4423**	**66**	**0.89**							
E_R_1(1000,0.27)	333	66.93	1,000	133,691	666	0.27	E_R_3(550,0.87)	452	27.84	550	131,744	366	0.87
E_R_1(150,0.86)	**118**	**7.53**	**150**	**9,629**	**100**	**0.86**	E_R_3(600,0.25)	199	30.91	600	44,216	400	0.25
E_R_1(200,0.82)	**147**	**10.20**	**200**	**16,398**	**133**	**0.82**	E_R_3(650,0.45)	246	46.14	650	94,737	433	0.45
E_R_1(250,0.89)	205	12.58	250	27,660	166	0.89	E_R_3(700,0.44)	278	55.91	700	107,843	466	0.44
E_R_1(300,0.34)	99	16.78	300	15,463	200	0.34	E_R_3(750,0.94)	676	38.09	750	263,341	500	0.94
E_R_1(350,0.32)	116	20.19	350	19,469	233	0.32	E_R_3(800,0.61)	437	65.80	800	194,491	533	0.61
E_R_1(400,0.79)	290	20.52	400	62,725	266	0.79	E_R_3(850,0.27)	283	84.29	850	96,923	566	0.27
E_R_1(450,0.25)	149	22.90	450	25,796	300	0.25	E_R_3(900,0.81)	686	47.64	900	328,739	600	0.81
E_R_1(500,0.95)	454	25.10	500	118,500	333	0.95	E_R_3(950,0.80)	726	55.15	950	362,764	633	0.8
E_R_1(550,0.64)	316	36.51	550	96,866	366	0.64	E_R_4(100,0.32)	33	5.01	100	1,521	66	0.32
E_R_1(600,0.59)	318	39.63	600	106,153	400	0.59	E_R_4(1000,0.55)	512	67.07	1, 000	276,724	666	0.55
E_R_1(650,0.24)	216	38.10	650	50,342	433	0.24	E_R_4(150,0.69)	89	7.60	150	7,766	100	0.69
E_R_1(700,0.41)	254	57.63	700	100,317	466	0.41	E_R_4(200,0.61)	102	10.42	200	12,162	133	0.61
E_R_1(750,0.57)	401	54.65	750	160,715	500	0.57	E_R_4(250,0.69)	153	12.76	250	21,437	166	0.69
E_R_1(800,0.31)	266	76.69	800	99,633	533	0.31	E_R_4(300,0.35)	99	16.59	300	15,636	200	0.35
E_R_1(850,0.91)	746	44.40	850	328,272	566	0.91	E_R_4(350,0.22)	116	18.77	350	13,506	233	0.22
E_R_1(900,0.37)	299	55.39	900	149,871	600	0.37	E_R_4(400,0.86)	307	20.18	400	68,674	266	0.86
E_R_1(950,0.81)	733	54.83	950	366,511	633	0.81	E_R_4(450,0.94)	407	22.68	450	95,316	300	0.94
E_R_2(100,0.12)	28	5.10	100	628	66	0.12	E_R_4(500,0.75)	344	27.09	500	93,264	333	0.75
E_R_2(1000,0.30)	333	69.41	1,000	150,791	666	0.3	E_R_4(550,0.83)	432	29.45	550	125,653	366	0.83
E_R_2(150,0.51)	60	7.87	150	5,697	100	0.51	E_R_4(600,0.76)	425	35.15	600	137,495	400	0.76
E_R_2(200,0.23)	65	10.83	200	4,555	133	0.23	E_R_4(650,0.59)	348	33.06	650	124,168	433	0.59
E_R_2(250,0.81)	183	12.81	250	25,140	166	0.81	E_R_4(700,0.62)	397	39.81	700	150,995	466	0.62

Table 9 continued

Instance	VS	Time (s)	n	m	b	prob.	Instance	VS	Time (s)	n	m	b	prob.
E_R_2(300,0.52)	132	15.14	300	23,196	200	0.52	E_R_4(750,0.57)	380	55.15	750	160,390	500	0.57
E_R_2(350,0.19)	115	20.72	350	11,621	233	0.19	E_R_4(800,0.98)	765	40.25	800	312,254	533	0.98
E_R_2(400,0.40)	133	25.61	400	32,249	266	0.4	E_R_4(850,0.74)	592	52.94	850	265,126	566	0.74
E_R_2(450,0.10)	145	29.39	450	10,034	300	0.1	E_R_4(900,0.35)	299	57.31	900	139,288	600	0.35
E_R_2(500,0.05)	155	34.93	500	6,340	333	0.05	E_R_4(950,0.45)	371	67.88	950	202,104	633	0.45
E_R_2(550,0.33)	183	36.16	550	49,185	366	0.33	**E_R_5(100,0.71)**	**58**	**5.00**	**100**	**3493**	**66**	**0.71**
E_R_2(600,0.21)	199	44.55	600	38,330	400	0.21	E_R_5(1000,0.86)	821	57.03	1, 000	427,339	666	0.86
E_R_2(650,0.36)	216	46.49	650	76,761	433	0.36	E_R_5(150,0.07)	42	8.25	150	852	100	0.07
E_R_2(700,0.49)	300	52.72	700	119,151	466	0.49	E_R_5(200,0.44)	69	10.82	200	8,784	133	0.44
E_R_2(750,0.94)	678	38.34	750	262,784	500	0.94	E_R_5(250,0.68)	149	12.99	250	21,241	166	0.68
E_R_2(800,0.36)	266	41.35	800	116,656	533	0.36	E_R_5(300,0.36)	99	16.77	300	16,225	200	0.36
E_R_2(850,0.64)	511	61.06	850	231,045	566	0.64	E_R_5(350,0.55)	172	20.44	350	33,598	233	0.55
E_R_2(900,0.61)	511	79.59	900	249,172	600	0.61	E_R_5(400,0.38)	133	25.18	400	30,202	266	0.38
E_R_2(950,0.83)	755	50.55	950	374,325	633	0.83	E_R_5(450,0.25)	149	22.94	450	25,629	300	0.25
E_R_3(100,0.78)	**61**	**5.01**	**100**	**3,856**	**66**	**0.78**	E_R_5(500,0.21)	165	30.53	500	26,000	333	0.21
E_R_3(1000,0.92)	891	52.16	1,000	459,891	666	0.92	E_R_5(550,0.60)	290	31.48	550	90,484	366	0.6
E_R_3(150,0.38)	49	7.88	150	4,358	100	0.38	E_R_5(600,0.24)	199	30.70	600	43,683	400	0.24
E_R_3(200,0.35)	66	10.75	200	6,980	133	0.35	E_R_5(650,0.65)	390	41.26	650	136,458	433	0.65
E_R_3(250,0.37)	83	14.10	250	11, 572	166	0.37	E_R_5(700,0.94)	633	35.29	700	230, 046	466	0.94
E_R_3(300,0.25)	99	15.33	300	11,480	200	0.25	E_R_5(750,0.70)	473	52.23	750	196,434	500	0.7
E_R_3(350,0.55)	161	18.25	350	33,481	233	0.55	E_R_5(800,0.38)	266	42.81	800	121,869	533	0.38
E_R_3(400,0.11)	130	22.92	400	9,135	266	0.11	E_R_5(850,0.33)	283	44.08	850	119,196	566	0.33
E_R_3(450,0.75)	309	22.98	450	76,297	300	0.75	E_R_5(900,0.22)	299	47.90	900	89,579	600	0.22
E_R_3(500,0.50)	223	36.00	500	62,072	333	0.5	E_R_5(950,0.29)	316	59.51	950	130,348	633	0.29

References

1. Arora, S., Rao, S., Vazirani, U.: Expander flows, geometric embeddings, and graph partitionings. In: 36th Annual Symposium on the Theory of Computing, pp. 222–231 (2004)
2. Bhatt, S.N., Leighton, F.T.: A framework for solving VLSI graph layout problems. J. Comput. Syst. Sci. **28**, 300–343 (1984)
3. Biha M.D., Meurs, M.J.: An exact algorithm for solving the vertex separator problem. J. Glob. Optim. **49**, 425–434 (2011)
4. Bui, T.N., Jones, C.: Finding good approximate vertex and edge partitions is NP-hard. Inf. Process. Lett. **42**, 153–159 (1992)
5. Duarte, A., Escudero, L.F., Martí, R., Mladenović, N., Pantrigo, J.J., Sánchez-Oro, J.: Variable neighborhood search for the vertex separation problem. Comput. Oper. Res. **39**(12), 3247–3255 (2012)
6. Feige, U., Kogan, S.: Hardness of approximation of the balanced complete bipartite subgraph problem. Technical report MCS04-04, Department of Computer Science and Applied Math., The Weizmann Institute of Science (2004)
7. Feige, U., Hajiaghayi, M., Lee, J.R.: Improved approximation algorithms for minimum-weight vertex separators. In: Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing, pp. 563–572 (2005)
8. Fu, B., Chen, Z.: Sublinear time width-bounded separators and their application to the protein side-chain packing problem. In: Algorithmic Aspects in Information and Management, LNCS, vol. 4041, pp. 149–160 (2006)
9. Hansen, P., Mladenović, N., Moreno, J.A.: Variable neighborhood search: methods and applications. Ann. Oper. Res. **175**(1), 367–407 (2010)
10. Leighton, F.T.: Complexity Issues in VLSI: Optimal Layout for the Shuffle-Exchange Graph and Other Networks. MIT Press, Cambridge (1983)
11. Leighton, T., Rao, S.: Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. J. ACM **46**, 787–832 (1999)
12. Leiserson, C.: Area-efficient graph layouts (for VLSI). In: 21th Annual Symposium on Foundations of Computer Science. IEEE Computer Soc., Los Alamitos, pp. 270–280 (1980)
13. Lipton, R.J., Tarjan, R.J.: A separator theorem for planar graphs. SIAM J. Appl. Math **36**, 177–189 (1979)
14. Lipton, R.J., Tarjan, R.E.: Applications of a planar separator theorem. SIAM J. Comput. **9**, 615–627 (1980)
15. Lozano, M., Duarte, A., Gortázar, F., Martí, R.: Variable neighborhood search with ejection chains for the antibandwidth problem. J. Heuristics **18**, 919–938 (2012)
16. Martí, R., Moreno-Vega, J.M., Duarte, A.: Advanced multi-start methods. In: Gendreau, M., Potvin, J-Y. (eds.) Handbook of Metaheuristics, vol. 146, pp. 265–281. Springer (2010)
17. Mladenović, N., Hansen, P.: Variable neighborhood search. Comput. Oper. Res. **24**, 1097–1100 (1997)
18. Montemayor, A.S., Duarte, A., Pantrigo, J.J., Cabido, R.: High-performance VNS for the Max-cut problem using commodity graphics hardware. In: Mini-Euro Conference on VNS (MECVNS 05), Tenerife (Spain), pp. 1–11 (2005)
19. Pardo, E.G., Mladenović, N., Pantrigo, J.J., Duarte, A.: Variable formulation search for the cutwidth minimization problem. Appl. Soft Comput. **13**, 2242–2252 (2013)
20. Sánchez-Oro, J., Pantrigo, J.J., Duarte, A.: Balancing intensification and diversification strategies in VNS. An application to the Vertex Separation Problem. Technical Report. Dept. Ciencias de la Computación. Universidad Rey Juan Carlos (2013)
21. de Souza, C., Balas, E.: The vertex separator problem: algorithms and computations. Math. Progr. **103**, 609–631 (2005)
22. Hansen, P., Mladenović, N.: Variable neighborhood search. Int. Ser. Oper. Res. Manag. Sci. **57**, 145–184 (2003)