Two-dimensional bandwidth minimization problem: exact and heuristic approaches

Miguel Ángel Rodríguez-García[a], Jesús Sánchez-Oro[a,*], Eduardo Rodriguez-Tello[b], Éric Monfroy[c], Abraham Duarte[a]

[a]*Dept. Computer Sciences, Universidad Rey Juan Carlos, Tulipán s/n, 28933 Móstoles (Madrid), Spain*
[b]*Cinvestav Tamaulipas. Km 5.5 Carretera Victoria-Soto La Marina, 87130 Victoria Tamps., Mexico*
[c]*LERIA, Université d'Angers. 2 Boulevard Lavoisier, 49045 Angers, France*

Abstract

Reducing the bandwidth in a graph is an optimization problem which has generated significant attention over time due to its practical application in Very Large Scale Integration (VLSI) layout designs, solving system of equations, or matrix decomposition, among others. The bandwidth problem is considered as a graph labeling problem where each vertex of the graph receives a unique label. The target consists in finding an embedding of the graph in a line, according to the labels assigned to each vertex, that minimizes the maximum distance between the labels of adjacent vertices. In this work, we are focused on a 2D variant where the graph has to be embedded in a two-dimensional grid instead. To solve it, we have designed two constructive and three local search methods which are integrated in a Basic Variable Neighborhood Search (BVNS) scheme. To assess their performance, we have designed three Constraint Satisfaction Problem (CSP) models. The experimental results show that our CSP models obtain remarkable results in small or medium size instances. On the other hand, BVNS is capable of reaching equal or similar results than the CSP models in a reduced run-time for small instances, and it can provide high quality solutions in those instances which are not optimally solvable with our CSP models.

Keywords: Graph layout, metaheuristics, Variable Neighborhood Search, GRASP, Constraint Programming Formulations

1. Introduction

The bandwidth minimization problem (BMP) and its variants have been extensively studied in the literature for their applications in the context of circuit layout, Very Large Scale Integration (VLSI) design, or network communications [1, 2, 3]. Formally, this family of problems is defined as follows. Let $H = (V_H, E_H)$

*Corresponding author

Email addresses: `miguel.rodriguez@urjc.es` (Miguel Ángel Rodríguez-García), `jesus.sanchezoro@urjc.es` (Jesús Sánchez-Oro), `ertello@cinvestav.mx` (Eduardo Rodriguez-Tello), `eric.monfroy@univ-angers.fr` (Éric Monfroy), `abraham.duarte@urjc.es` (Abraham Duarte)

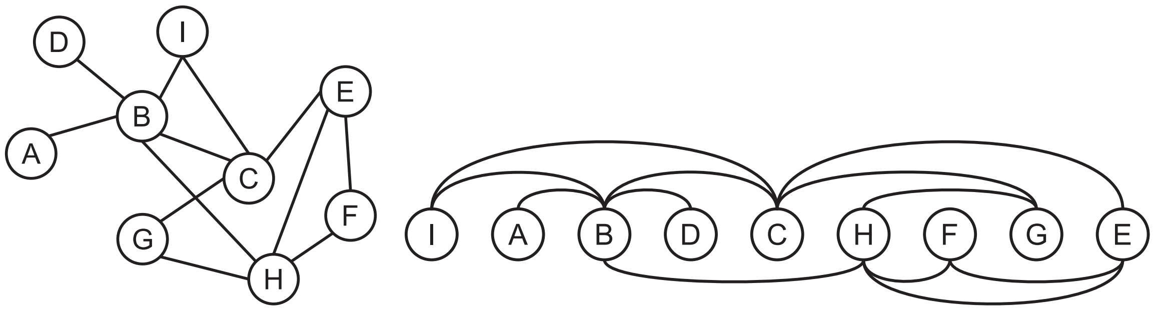
and $G = (V_G, E_G)$ be a host graph and a guest graph, respectively, and let φ be an injective function (usually known as labeling or embedding) of the guest graph to the host graph, i.e., $\varphi : V_G \longrightarrow V_H$. Then, the bandwidth (cost) of this labeling is computed as:

$$\mathrm{B}(G, \varphi) = \max_{(u,v) \in E_G} d_H(\varphi(u), \varphi(v)) \ , \tag{1}$$

where the function $d_H(x, y)$ computes the distance between x and y in H. The optimal bandwidth of G, relative to the host graph H, is then defined as the minimum $\mathrm{B}(G, \varphi)$ value considering the set of all possible labelings Φ of G. In mathematical terms,

$$\mathrm{B}^\star(G) = \min_{\varphi \in \Phi} \mathrm{B}(G, \varphi) \ . \tag{2}$$

The most studied variant of this family of problems, is called linear bandwidth (LBMP), and considers that H is a path graph of order $n = |V_H| = |V_G|$, which is usually represented as a horizontal line. In this case, the embedding φ of G to H consists in assigning the integers $\{1, 2, \ldots, n\}$, where $\varphi(u) = i$ indicates that vertex $u \in V_G$ is located in position i in the line, with $1 \leq i \leq n$. Figures 1(a) and 1(b) show an example graph G with 5 vertices and 7 edges and a feasible embedding in a path graph, respectively. As it can be observed, $\varphi(\mathtt{I}) = 1$ since vertex `I` is located in the first position in the line. Similarly, $\varphi(\mathtt{C}) = 5$ which means that vertex `C` is located in the fifth position in the line. The linear bandwidth of the labeling depicted in Figure 1(b) is determined by the maximum distance in the host graph between adjacent vertices (in the guest graph), i.e., $\mathrm{B}_P(\varphi) = 4$. It corresponds to the distance between vertices `I` and `C` in the host graph.



(a) Example graph with 5 vertices and 7 edges.

(b) Example labeling of the graph.

Figure 1: Undirected graph G and a possible labeling of G for the linear bandwidth minimization problem.

This variant of the bandwidth problem was proven to be $\mathcal{NP}$-complete [4] and, since then, it is an optimization problem which has been extensively studied. The scientific community has approached this problem by considering two lines of research. The first one is devoted to deriving theoretical results (mainly lower bounds) and exact methods for the linear bandwidth. Specifically, Chvaťal [5] proposed a lower bound based on the density of the graph. Gurari and Sudborough [6] designed an improved dynamic programming algorithm which solves the LBMP in $O(n^b)$ steps, b being the searched bandwidth. Del Corso and Manzini [7] proposed an exact procedure based on the branch-and-cut methodology to solve small and medium instances (up to 100 vertices). Caprara and Salazar [8], extend the previous one by introducing tighter lower bounds, thus solving large size instances. Martí et al. [9] introduced a branch-and-bound coupled with a metaheuristic procedure that outperformed all previous approaches.

The second line of research is devoted to heuristic approaches. For many years, researchers were only interested in designing relatively simple heuristic procedures, which were extremely fast but, unfortunately, the quality of the obtained solutions were poor. See for instance [10, 11]. In the last 20 years, researches have focused on proposing advanced metaheuristics which are able to find high-quality solutions in moderate computing time. A tabu search is presented in [12], where the authors reinterpret the linear bandwidth as a labelling problem. To solve it, the authors introduced a move strategy that reduces the bandwidth by swapping labels between vertices. The tabu procedure is designed through a short-term memory implementation which restricts the node re-tagging when it has been moved. Using the same move operator, Piñana et al. [13] proposed a Greedy Randomized Adaptive Search Procedure (GRASP) in combination with Path Relinking to improve the performance compared to the tabu search. Conversely, from a different perspective, we can also find trajectory-based metaheuristics based on soft computing methodologies. For instance, Lim et al. [14] proposed a hybrid strategy based on ant colony and hill-climbing or Czibula et al. [15] proposed a soft computing methodology which consisted on combining a genetic algorithm and an ant-based system to resolve the bandwidth problem. Rodriguez-Tello et al. [16] proposed a simulated annealing procedure with an alternative objective function to efficiently explore the search space. Mladenovic et al. [17] introduced a Variable Neighborhood Search (VNS) heuristic. Their implementation considered a reduced swap neighborhood, a fast local search procedure, and a rotation-based neighborhood structure. Finally, Torres-Jimenez et al. [18] reported a dual representation simulated annealing algorithm based on a compound neighborhood function to provide high quality solutions to the LBMP. As far as we know, this method is currently considered the state of the art in this optimization problem.

In this paper, we focus on the two-dimensional bandwidth minimization problem (*2DBMP*), originally proposed in [19]. This variant considers the embedding of a guest graph $G = (V_G, E_G)$ into a host graph $G_H = (V_H, E_H)$, which is a two-dimensional grid with size $n \times n$, being $|V_H| = |V_G| = n$. In this case, the embedding φ of G to H assigns a 2-tuple to each vertex that corresponds with the row and column of the grid node[1] where the vertex is located. Specifically, $\varphi(u) = (i, j)$ indicates that vertex $u \in V_G$ is located in the grid node at row i and column j, with $1 \leq i, j \leq n$.

[1]Hereafter, the term *node* is used to refer to vertices in the grid host, while *vertex* is employed in the context of the guest graph.

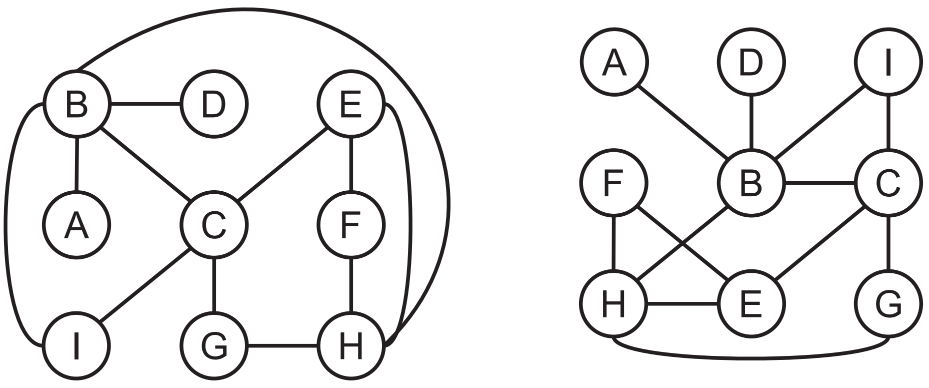
As it was aforementioned, the distance function strongly depends on the host graph. Considering that *2DBMP* was originally derived from circuit layout models (where wires are traced in horizontal/vertical directions), this metric usually represents the rectilinear distance computed with the L_1-norm distance [20]. Therefore, given two vertices $u, v \in V_G$ and a labeling φ such that $\varphi(u) = (i, j)$ and $\varphi(v) = (i', j')$, the distance between them in the grid is:

$$d_{L_1}(\varphi(u), \varphi(v)) = d_{L_1}((i, j), (i', j')) = |i - i'| + |j - j'| \tag{3}$$

Therefore, Equation 1 is particularized for the *2DBMP* by considering L_1-norm instead of a general distance function (defined with the d_H function) as follows:

$$\mathrm{B}_{2D}(G, \varphi) = \max_{(u,v)\in E_G} d_{L_1}(\varphi(u), \varphi(v)) \ , \tag{4}$$

and it represents the objective function of the *2DBMP*.



(a) Labeling φ_1. (b) Labeling φ_2.

Figure 2: Example of two different labelings for the graph depicted in Figure 1(a).

Figures 2(a) and 2(b) show two possible labelings, φ_1 and φ_2, for the graph depicted in Figure 1(a). As it can be observed in Figure 2(a), $\varphi_1(\mathtt{B}) = (1, 1)$ since vertex B is located in the first row and first column of the grid. Similarly, $\varphi_2(\mathtt{C}) = (2, 3)$, which means that vertex C is located in the second row and third column of the grid in φ_2 (see Figure 2(b)).

Considering the embedding φ_1 of the guest example graph (see Figure 1(a)) to a 3×3 bi-dimensional grid, the largest distance between any pair of vertices is obtained when considering B and H. In particular, $d_{L_1}(\varphi_1(\mathtt{B}), \varphi_1(\mathtt{H})) = d_{L_1}((1, 1), (3, 3)) = |1-3|+|1-3| = 4$. If we now consider φ_2, we can see that the maximum distance corresponds to the one between vertices H and G, so $d_{L_1}(\varphi_2(\mathtt{G}), \varphi_2(\mathtt{H})) = d_{L_1}((1, 3), (3, 3)) =$

$|1-3| + |3-3| = 2$. Thus, we can conclude that φ_2 is a better solution than φ_1 for the *2DBMP* since it presents a smaller objective function value.

The *2DBMP* has been also proven to be $\mathcal{NP}$-complete [21, 22]. This problem is useful for optimizing the design of VLSI [2], in order to generate more efficient circuits, in terms of area and wire lengths. *2DBMP* can be also applied for optimizing the communication requirements in grid-based parallel computers, which are usually considered when parallelizing divide-and-conquer algorithms [1]. If we model parallel algorithms or massively parallel computers as graphs, where vertices are processes or processor units, and edges indicate the relations among them, the problem of efficiently executing a parallel algorithm on a parallel computer is reduced to mapping the graph that represents the algorithm on the graph that represents the parallel computer, satisfying the required constraints. As stated in [3], the *2DBMP* can be useful for generating those embedding.

Despite the high number of applications of the *2DBMP*, this problem have been mainly approached by proposing lower bounds for general graphs [20, 23, 24]. To the best of our knowledge, there are not previous algorithms for solving the *2DBMP* neither from an exact nor from a heuristic perspective.

The main contributions of this paper are related with the introduction of new solving procedures for the *2DBMP* that can be easily adapted for dealing with other hard optimization problems. Specifically, a Constraint Satisfaction Problem (CSP) model, named M_0, to optimally solve small and medium size instances. We propose three refinements of M_0 denoted as M_1, M_2, and M_3, where each model incrementally includes the strategies of the previous one. The first model is intended to avoid costly conjunctions; the second model additionally includes global constraints to increase the efficiency of the solver; finally, the third model also incorporates strategies to prevent exploring symmetric solutions in the search space. We then propose a Variable Neighborhood Search (VNS) algorithm to deal with large size instances. This metaheuristic is composed of an original constructive procedure and six novel local search procedures designed for exploring a large portion of the search space. These local search methods are based on three exploration strategies: first improvement (FI), best improvement (BI), and a novel hybrid approach which leverages the best of both FI and BI.

The remaining of the paper is organized as follows. Section 2 presents the Constraint Satisfaction Problem models designed for finding optimal solutions for the *2DBMP*. Section 3 proposes a metaheuristic algorithm for solving large-scale instances efficiently. Section 4 evaluates the presented algorithms in terms of quality and computing times, showing the limitations of exact and heuristic approaches. Finally, Section 5 draws the conclusions derived from this research.

2. Constraint Programming Formulations for the *2DBMP*

In order to obtain a clear basis, we need a sharp separation between the optimization problem and its resolution. To this end, we consider a declarative programming paradigm which provides a high-level language for modeling our problem and a generic method for solving it. On the one hand, the probabilistic programming approaches [25] and probabilistic graphical models provide a formal language for modeling and, additionally, a common target for efficient inference algorithms. Probabilistic programming languages aim at unifying general purpose programming with probabilistic modeling: the user specifies a probabilistic model and inference follows automatically given the corresponding specification. On the other hand, constraint programming [26] is originated from constraint logic programming [27]. It is a paradigm for solving combinatorial problems that also uses techniques from Artificial Intelligence and Computer Science mainly. In constraint programming, the user just declaratively states what her/his problem is, and then the computer automatically solves it. Constraint programming differs from usual programming languages in that it does not require to specify steps or sequences of steps to execute, but rather properties of solutions to be found.

Both of these programming approaches present a clean dissociation from modeling and reasoning/inference. However, since we do not have to handle uncertainty in our data (an instance is directly determined with a graph), and that we require numerous arithmetic operations to compute and compare bandwidths, we turn towards constraint programming.

In constraint programming, stating the problem or *modeling the problem* consists in defining a set of objects that must satisfy some constraints or limitations. Practically, the model or the constraint satisfaction problem is given by a set of decision variables with their domains (the candidate values of these variables), together with some relations (the constraints) linking these variables. A solution is a tuple of the search space (defined by the Cartesian product of the domains of variables) which satisfies all the constraints. The user may also need a solution that optimizes some criteria. In this case, it is usually known as constraint optimization problem. In this paper, we are concerned with a constraint optimization problem, i.e., the *2DBMP*, which requires finding the best solution (the solution which minimizes the maximum distance between labels) which satisfies some constraints (vertices are labeled with different nodes of the grid). A model together with the required data conforms an instance, which is solved with an algorithm that we call a *constraint solver*.

There are numerous possible methods and algorithms that can be solvers, issued from Artificial Intelligence, Mathematical Programming, Metaheuristics, Operational Research, etc. Most of the time, the solvers are constraint-propagation-based solvers. This kind of solvers is said to be complete: if there is no solution, they prove it; if there is a solution, they find it. In case of optimization, they find the global optimal solution if they are given sufficient time.

Constraint-propagation-based solvers combine computation and deduction. They are based on an iterative process that alternates constraint propagation and search. Constraint propagation consists in reducing the search space by removing those values (from domains of variables) that cannot participate in any solution. Search consists in branching in the search tree by enumerating a variable. Completeness is assured by means of backtracking strategies.

In the following, we will thus focus on modeling the *2DBMP*. Modeling is also an iterative process which consists in producing a first model which is then refined in order to be solved more efficiently. Modeling consists in defining decision variables together with their domains (candidate values), and some constraints linking these variables. Refining the model may be performed by rewriting some constraints in a better-suited way for the solver, adding some redundant constraint that simplifies and speeds-up the solving process, using some global constraints, breaking some symmetries for reducing the search space, etc.

2.1. First Constraint Satisfaction Problem model

A Constraint Satisfaction Problem (CSP) model is usually determined by defining some data, variables (with their corresponding domains), and constraints (linking these variables). For the *2DBMP*, initial graphs are our **data**. For the sake of convenience, we represent the graph by considering a 2-dimensional array of edges, named as $graph(1..m, 1..2)$, where m indicates the number of edges of the guest graph. In this particular representation, $graph(i, 1)$ and $graph(i, 2)$ determines both extremes of the i-th edge of the graph. We additionally consider upper and lower bounds for the *2DBMP*, denoted as ub and lb, respectively, computed according to the following expressions proposed in [24]:

$$lb = \frac{\delta(n)}{D(G)}\ , \tag{5}$$

and

$$ub = \delta(n)\ , \tag{6}$$

where $D(G)$ represents the diameter of the graph G, n is the number of vertices of the host/guest graphs, and $\delta(n)$ is the minimal diameter for a set of n grid points calculated as follows:

$$\delta(n) = \min\left\{2\left\lceil\frac{\sqrt{2n-1}-1}{2}\right\rceil,\ 2\left\lceil\sqrt{\frac{n}{2}}\right\rceil - 1\right\}\ . \tag{7}$$

The reader is referred to Appendix A and Appendix B for consulting a list of graphs with known bounds, and a collection of graphs with known optimal solutions, respectively.

The objective function depicted in Equation 4 is implemented in the model to evaluate the feasible CSP states.

The second stage in the CSP model consists in defining the decision **variables** (or variables, for short), that we need in our model. Specifically, they are meant to capture distances between vertices of every edge and labels assigned to vertices. Distance variables are stored in an array of variables ranging from the minimum distance to the maximum distance. More precisely, it is named as $distance(1..m)$ where $distance(i)$ is a variable ranging from 1 to ub that represents the distance between the two vertices of edge i. Labels of the n vertices are stored in a 2-dimensional array of grid coordinates. In particular, $grid(1..n, 1..2)$ is an array of labels, where $grid(i, 1)$ and $grid(i, 2)$ indicates the row and column where the i-th vertex of the guest graph is located in the host graph.

Constraints in the CSP model for the *2DBMP* are focused on distances and uniqueness of labels. Specifically, distance constraints exactly correspond to the definition of distance given in Section 1 (see Equation 1):

$$\begin{aligned} & \forall i \in [1..m] \\ distance(i) \;=\; & |grid(graph(i,2),1) - grid(graph(i,1),1)| + \\ & |grid(graph(i,2),2) - grid(graph(i,1),2)| \;. \end{aligned} \tag{8}$$

Similarly, constraints about labels establish that two vertices of the guest graph cannot correspond to the same vertex of the grid:

$$\begin{gathered} \forall v_1 \in [1..n-1],\ \forall v_2 \in [v_1+1..n] \\ grid(v_1,1) \neq grid(v_2,1) \ \vee \ grid(v_1,2) \neq grid(v_2,2) \;. \end{gathered} \tag{9}$$

These constraints state that if two vertices are on the same row, they must be on different columns of the grid.

The *2DBMP* is an optimization problem, therefore, we need to state which variable must be optimized. Specifically, the maximum of the distances is defined as follows (see Equation 1, where the distance function es particularized in Equation 3):

$$max\_distance = \max\{distance(i) | i \in [1..m]\} \;. \tag{10}$$

Then, according to the definition introduced in Section1, the *2DBMP* requires to minimize the maximum distance (see Equation 2):

$$minimize(max\_distance) \;. \tag{11}$$

Finally, the first model for the *2DBMP*, denoted as M_0, is given by the variable definition and the following conjunction of constraints and objective:

$$M_0 = (9) \wedge (8) \wedge (10) \wedge (11) \;.$$

2.2. Refined models

Programming is an iterative process which consists in coding a first version of the program and then refining it to improve its readability or efficiency. Modeling is also an iterative process which consists in refining models in order to improve the solving efficiency of the model. This can be done by adding redundant constraints, i.e., constraints that do not change neither the semantics of the model nor its solutions, but that can help the solver. Another method consists in changing a part of the model by some constraints that are solved more efficiently. In this section, we propose three refinements (M_1, M_2, and M_3) based on the original one, where we explore some strategies to improve its effectiveness.

In the first model, M_1, we try to avoid disjunctions since they are generally costly to be solved in constraint programming. As can be seen above, Constraints (9) introduce some disjunctions. We then refine the aforementioned model, M_0, by replacing these costly constraints by some others that can be evaluated in a more efficient way. In particular, we determine that two vertices of the guest graph cannot correspond to the same vertex of the host graph (grid), as follows:

$$\begin{gathered}\forall i \in [1..n-1], \forall j \in [i+1..n], \\ grid(i,1) * \lceil \sqrt{n} \rceil + grid(i,2) \neq \\ grid(j,1) * \lceil \sqrt{n} \rceil + grid(j,2)\ .\end{gathered} \tag{12}$$

Constraints (12) mean that two vertices i and j are not on the same node of the grid. Constraints (12) replace Constraints (9). Note that (12) consist of $n(n-1)/2$ inequalities. Therefore, the model M_1 is thus given by: $M_1 = (12) \wedge (8) \wedge (10) \wedge (11)$.

In the second refinement, M_2, we include global constraints that can be stated as the conjunction of several "basic" constraints. Global constraints are thus "syntactic sugar" that simplifies modeling and readability. Moreover, some specific algorithms are dedicated to the global constraints, and thus, they improve the efficiency of solvers. More precisely, a global constraint captures a relation among n variables, n not being fixed. One of the most famous global constraint is the $\mathit{all\_different}(x_1, \ldots, x_n)$ constraint, which specifies that the values of the variables $x_1, \ldots, x_n$ must be pairwise distinct, i.e., $\forall i \in [1..n-1], \forall j \in [i+1..n], x_i \neq x_j$.

Generally speaking, solvers take advantage of the structure of part of the problem modeled with global constraints. Then, we improve model M_1 using one *all_different* global constraint by considering that two vertices of the guest graph cannot correspond to the same vertex of the grid:

$$\mathit{all\_different}(\{grid(i,1) * \lceil \sqrt{n} \rceil + grid(i,2) \mid i \in [1..n]\})\ . \tag{13}$$

Constraints (13) mean that two vertices are not on the same node of the grid using the *all_different* global constraint. Constraints (13) replace the $n(n-1)/2$ Constraints (12). Then, the model M_2 is given by: $M_2 = (13) \wedge (8) \wedge (10) \wedge (11)$.

In the third refinement, M_3, we consider specific strategies for symmetry breaking. Symmetries occur in many constraint satisfaction or constraint optimization problems. It is thus crucial to deal with symmetries or the solver will waste much time exploring solutions and branches of the search tree which are symmetric to already explored parts. A usual method to take advantage of symmetries [28] consists in complementing the model with constraints [29] that eliminate isomorphic solutions, and consequently symmetries of the search space.

The *2DBMP* presents obvious vertical and horizontal symmetries. It is thus possible to place the first vertex on the lower left quarter of the grid. Hence, only symmetric solutions are lost, and note that these "lost" solutions can be deduced from the computed solutions. The search space is significantly reduced, and the model is solved more efficiently. We then consider that the first vertex is placed on the lower left quarter of the grid:

$$grid(i,1) \leq \lceil \sqrt{n} \rceil \div 2 \quad \wedge \quad grid(i,2) \leq \lceil \sqrt{n} \rceil \div 2 \, . \tag{14}$$

The refined model M_3 with symmetry breaking is given by: $M_3 = M_2 \wedge (14)$.

3. Heuristic approach

The main advantage of exact methods such as the CSP models, presented in Section 2, is that they are able to provide the optimal value for a given instance of *2DBMP*. However, their main drawback resides in the computational effort, in terms of both computing time and memory usage, required for solving a specific instance. Therefore, these algorithms are not applicable to medium or large instances. We then additionally introduce a metaheuristic algorithm based on Variable Neighborhood Search (VNS). This methodology was designed for solving hard optimization problems, and its potential lies on systematic changes of neighborhood structures. The VNS framework has been widely studied in the scientific community, resulting in a wide variety of VNS strategies: Reduced VNS (RVNS), Variable Neighborhood Descent (VND), Basic VNS (BVNS), General VNS (GVNS), among others. See [30] for a complete survey on VNS variants. This paper is focused on BVNS, since it offers extraordinary chances to move through neighborhoods, combining deterministic and stochastic changes.

BVNS requires three input parameters: the graph to be solved, G; the initial solution in which the search starts, φ; and the maximum neighborhood to be explored, $k_{\max}$. In the framework of VNS, the initial solution can be generated either at random or by using a heuristic procedure. In order to start the search from a high quality solution, we consider the latter since it usually produces better outcomes [31, 32]. See Section 3.1 for a further description.

We show in Algorithm 1 the pseucocode for BVNS. It starts the search from the first neighborhood to be explored, $k = 1$ (step 1). Then, BVNS iterates until reaching the maximum predefined neighborhood

Algorithm 1 *BVNS*$(G = (V, E), \varphi, k_{\max})$

```
1: $k = 1$
2: while $k \leq k_{\max}$ do
3:     $\varphi' \leftarrow Shake(\varphi, k)$
4:     $\varphi'' \leftarrow LS(\varphi')$
5:     $k \leftarrow NeighborhoodChange(\varphi, \varphi'', k)$
6: end while
7: return $\varphi$
```

$k_{\max}$ (steps 2-6). Each iteration firstly perturbs the incumbent solution φ with the *Shake* method (step 3), producing a random neighbor solution φ' in the current neighborhood k (see Section 3.4). Then, an improvement procedure, *LS*, is applied for finding a local optimum (step 4) producing a new solution φ'' (see Section 4). Finally, the *NeighborhoodChange* method selects the next neighborhood to be explored by considering the initial and final solution obtained in the current iteration, following the rules presented in Section 3.5. BVNS ends when no improvement is found in any of the neighborhoods, returning the best solution found during the search.

3.1. Initial solution

We propose two constructive procedures based on the GRASP methodology [33]. Specifically, instead of constructing a totally greedy solution, a controlled degree of randomization is considered to favor the diversification of whole procedure. Both methods start from an empty solution and label a vertex in each iteration. The first one considers the objective function as a criterion to guide the construction. Before defining this algorithm, we need to introduce the two-dimensional bandwidth of a specific vertex when not all vertices have been already labeled. In particular, let L and U be the set of labeled and unlabeled vertices, respectively. Notice that $L \cup U = V_G$ and $L \cap U = \emptyset$. We then define the two-dimensional bandwidth (cost) of a vertex v of the guest graph G with respect to φ as:

$$\mathrm{B}_{2D}(v, \varphi, G) = \min_{\substack{(u,v) \in E_G \wedge \\ u \in L \wedge v \in U}} d_{L_1}(\varphi(v), \varphi(u)) \ . \tag{15}$$

This equation evaluates the variation in the cost of a partial solution when vertex $v \in U$ is labeled with $\varphi(v)$ by considering only the adjacent vertices already labeled (those in L). As it was aforementioned, the label indicates the row and column (i, j) where v is embedded in the host graph. We propose a constructive procedure which only considers positions in H close to those already labeled. Actually, we define the set of available positions after labeling a vertex as those which differ in a single row or column (i.e., the adjacent ones in the host graph). More formally:

$$AP(v) = \{w \in V_H : d_{L_1}(\varphi(w), \varphi(v)) = 1\} \ . \tag{16}$$

Then, given a partial solution, the set of available positions is defined as:

$$AP = \bigcup_{v \in U} AP(v) \setminus \bigcup_{u \in L} \varphi(u) \ . \tag{17}$$

Notice that AP contains all positions in H adjacent to a position already labeled.

Once we have defined the set of available positions, we need to select which vertex will be placed in that position. Specifically, we compute the best position for an unlabeled vertex $v \in U$ as:

$$BestPos(v) = \operatorname*{arg\,min}_{\varphi(v) \in AP} \mathrm{B}_{2D}(v, \varphi, G) \ . \tag{18}$$

As it is well-documented in the related literature of labeling problems, it is really convenient to label first those vertices that are adjacent to an already labeled one [34, 35]. More formally, the set of adjacent vertices to a labeled vertex $v \in L$ is:

$$AV(v) = \{u \in V_G : (u, v) \in E_G, v \in L\} \ . \tag{19}$$

We therefore define the set of available vertices to be considered for labeling as:

$$AV = \bigcup_{v \in L} AV(v) \setminus L \ . \tag{20}$$

Having defined the set of available positions and available vertices, we propose a greedy function g_1 that evaluates the cost when a vertex $v \in U$ is labeled (located) in $BestPos(v)$. For each $v \in AV$, g_1 is computed as:

$$g_1(v) = \min_{\varphi(v) \in AP} \mathrm{B}_{2D}(v, \varphi, G) \ . \tag{21}$$

The first GRASP-based constructive procedure for *2DBMP* starts from an empty solution and, iteratively, adds new vertices to the incumbent solution until it becomes feasible (i.e., every vertex of V_G has been labeled with a different 2-tuple). Algorithm 2 shows the pseudocode of this method. The first vertex to be included in the solution is selected at random among the set of all vertices in the graph (step 1). Then, the i, j elements of a 2-tuple determining the position of the node in the two-dimensional grid are also selected at random in the range $1 \leq i, j \leq \lceil \sqrt{n} \rceil$. Remark that we decided to use a reduced grid with size $\lceil \sqrt{n} \rceil \times \lceil \sqrt{n} \rceil$, given the upper bound (see Equation 6) proposed in [24]. The corresponding vertex v is included in the set of labelled vertices (step 3). Next, the set of available positions and vertices are initialized (steps 4 and 5). Then, the constructive procedure iterates until including all the vertices in the incumbent solution (steps 6-16) returning, finally, the constructed solution (step 17). In each iteration, the constructive method determines the set AV of candidate vertices and the set AP of available positions where a vertex can be placed. Then, for each candidate vertex $v \in AV$, it computes the $g_1(v)$ that would result if the v was placed on each available position. Next, the algorithm considers these values to compute

g_{min} and g_{max} (step 6). After, g_{min} and g_{max} are used to calculate the threshold (step 10) and the corresponding Restricted Candidate List, RCL (step 10). The α parameter balances the greediness/randomness of the algorithm, where $\alpha = 0$ configures the algorithm to generate completely greedy solutions, while $\alpha = 1$ produces totally random solutions. Finally, the algorithm utilises the RCL to randomly select a candidate which is assigned to the best available position (step 12), updating the sets accordingly (steps 13 to 15).

Algorithm 2 $Constructive(G = (V, E), \alpha)$

```
1:  $v \leftarrow Random(V_G)$
2:  $\varphi(v) \leftarrow \left(Random(\lceil\sqrt{n}\rceil), Random(\lceil\sqrt{n}\rceil)\right)$
3:  $L \leftarrow \{v\}$
4:  $AP \leftarrow AP(v)$
5:  $AV \leftarrow AV(v)$
6:  while $AV \neq \emptyset$ do
7:      $g_{min} \leftarrow min_{v \in AV} g_1(v)$
8:      $g_{max} \leftarrow max_{v \in AV} g_1(v)$
9:      $Th \leftarrow (g_{min} + \alpha(g_{max} - g_{min}))$
10:     $RCL \leftarrow \{v \in AV : g_1(v) \leq Th\}$
11:     $u \leftarrow Random(RCL)$
12:     $\varphi(u) \leftarrow BestPos(u)$
13:     $AP \leftarrow AP \cup AP(u) \setminus \bigcup_{w \in L} \varphi(w)$
14:     $AV \leftarrow AV \cup AV(u) \setminus L$
15:     $L \leftarrow L \cup \{u\}$
16: end while
17: return $\varphi$
```

The second greedy function follows the idea proposed by McAllister [36] and later adapted by Pantrigo et al. [37] for labeling problems. Specifically, this greedy function evaluates the relevance of inserting a vertex as the number of labeled adjacent vertices minus the number of non-labeled adjacent vertices. More formally,

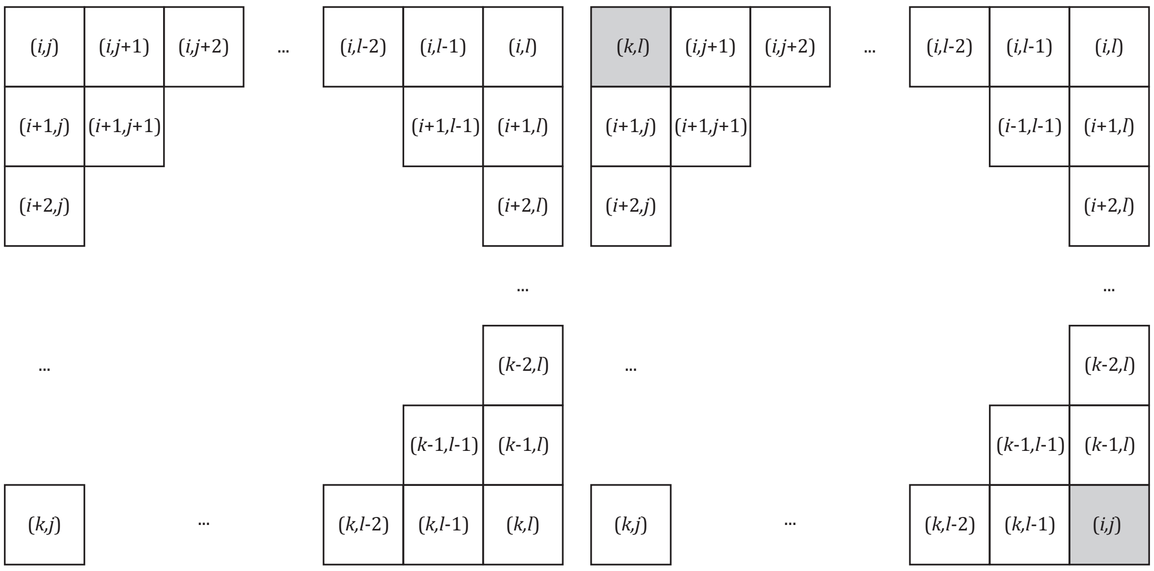
$$g_2(v) = |\{u \in AV(v) \cap L\}| - |\{u \in AV(v) \cap U\}| \tag{22}$$

For the sake of brevity we omit the inclusion of a pseudocode of this method since it is equivalent to the one presented in Algorithm 2, but substituting g_1 with g_2. We will study the experimental performance of these two methods in Section 4.2.

3.2. Neighborhood structures

Before defining the local search methods proposed for further improving the solutions constructed with the method described in Section 3.1, it is necessary to define the neighborhood structures considered for this problem. The neighborhood of a given solution φ is defined as the set of solutions that can be reached by performing a single move in φ. Therefore, the neighborhood structure highly depends on the move considered. We propose two different moves for the *2DBMP*: exchange and insert.

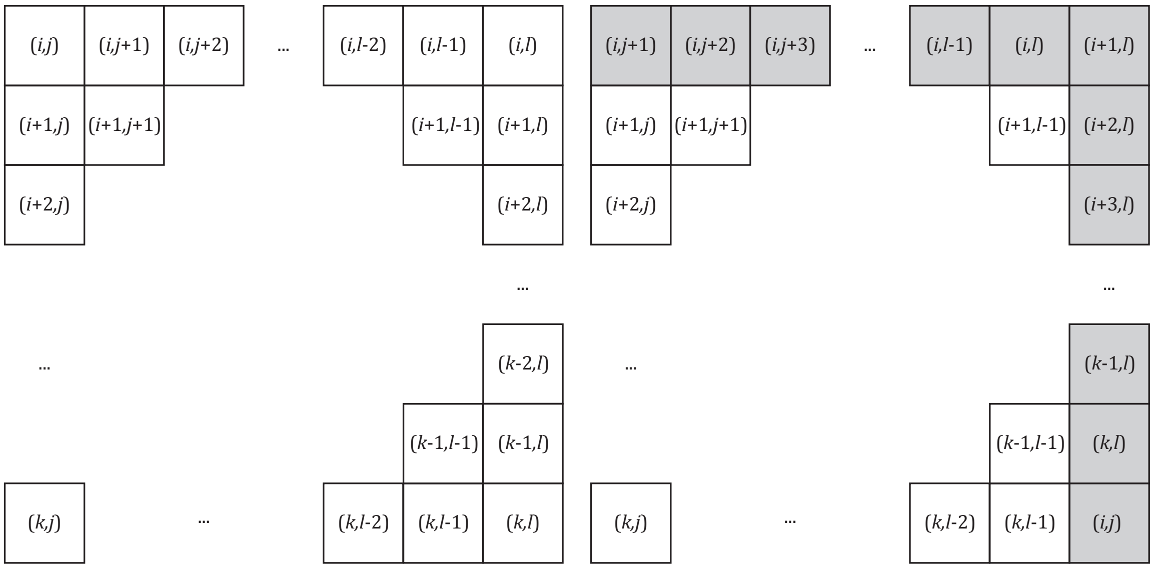
The exchange of two vertices u and v in a solution φ, defined as $Exc(\varphi, u, v)$ results in a new solution φ' where vertex u is labeled as $\varphi(v)$ and vertex v is labeled as $\varphi(u)$. Let us illustrate the move with a graphical example. Figure 3(a) shows the initial solution φ. Notice that we only show the region of the grid located between the nodes involved in the move. For the sake of simplicity, we will consider $1 \leq i \leq k \leq \lceil \sqrt{n} \rceil$ and $1 \leq j \leq l \leq \lceil \sqrt{n} \rceil$. Then, the result of performing $Exc(\varphi, u, v)$, with $\varphi(u) = (i, j)$ and $\varphi(v) = (k, l)$ is depicted in Figure 3(b), where the vertices whose label is modified during the move are highlighted in grey. In the case of the exchange move, only vertices located at position (i, j) and (k, l) have changed their labels.



(a) Initial solution. (b) Exchange move.

Figure 3: Definition of the exchange move, where the vertices whose labels are modified during the move are highlighted in grey.

The insertion of a vertex u, with label $\varphi(u) = (i, j)$, in the position assigned to another vertex v, located at $\varphi(v) = (k, l)$, named as $Ins(\varphi, u, v)$ consists in exchanging vertex u with the vertex located in the next column (i.e., $j + 1$) until reaching the column where v is located, i.e., until $\varphi(u) = (i, l)$. Then, u is interchanged with the vertices in the next row (i.e., $i+1$) until reaching the row where v is located, i.e., until $\varphi(u) = (k, l)$. In other words, vertex u is moved horizontally until reaching the column of v and, then, u is moved vertically until reaching the row of v, where the insert move ends. Figure 4(a) again shows the initial solution before performing the move. The result of performing the move $Ins(\varphi, u, v)$, with $\varphi(u) = (i, j)$ and $\varphi(v) = (k, l)$ is shown in Figure 4(b), where the vertices whose labels were modified are highlighted in grey.



(a) Initial solution. (b) Insert move.

Figure 4: Definition of the insertion move, where the vertices whose label is changed during the move are highlighted in grey.

As it can be derived from the figure, the label $\varphi(w)$ of a vertex w is modified by the insert move if and only if it is located in the same row than u and in a column y between j and l, i.e., those with a label $\varphi(w) = (i, y)$ with $j \leq y \leq l$, or if it is located in the same column than v and in a row x between i and k, i.e., those with a label $\varphi(w) = (x, l)$ with $i \leq x \leq k$.

Given the definition of these moves, we define two neighborhoods. On the one hand, the neighborhood $N_{Exc}(\varphi)$ contains all the solutions that can be reached from φ by performing a single exchange move. On the other hand, neighborhood $N_{Ins}(\varphi)$ is conformed with all the solutions derived from φ when performing a single insert move. More precisely,

$$N_{Exc}(\varphi) = \{\varphi' \leftarrow Exc(\varphi, u, v)\ \forall u, v \in V_G, u \neq v\}\ ,$$
$$N_{Ins}(\varphi) = \{\varphi' \leftarrow Ins(\varphi, u, v)\ \forall u, v \in V_G, u \neq v\}\ .$$

3.3. Local Search

Having defined the neighborhood structures, it is necessary to present the local search methods proposed to traverse the aforementioned neighborhoods $N_{Exc}(\cdot)$ and $N_{Ins}(\cdot)$. Thus, taking into account the way that the neighborhood structure is traversed and the moment when the improved solution is picked during the search, we have developed three different strategies. Thus, we have First Improvement (FI), Best Improvement (BI), and a combination of both, Hybrid first-best Improvement (HI) approaches. The strategies end when crossing over the neighborhood until no improving solution is found. In Section 4, we will discuss the performance of each local search method proposed.

The First Improvement (FI) strategy accepts the first solution in the neighborhood that results in improvement with respect to the objective function. Therefore, given a solution, the procedure iterates through its neighborhood to select a vertex. Then, it applies an exchanging or inserting movement depending on the operator provided. Next, if the selected solution improves the one given, the strategy will choose it as a current solution, and the search process will start again. As we can see, the order has a significant influence on the effectiveness of this strategy, since if the order in which the vertices are selected is not changed, the search will always explore the same first solutions, thus the search will be biased. A clear example would be if the strategy selects the vertices following a lexicographical ordering. It will explore the neighborhood in that order reducing the search diversification and provoking that the same first solutions will be selected. However, to solve this problem, we have randomized the vertices selection process to improve the diversification of the search process. In the Best Improvement (BI), unlike the FI strategy, the traversal order is not relevant because this strategy explores the complete neighborhood in each search step selecting the solution which obtained the best objective function value. Hence, from an initial solution, the local search iterates through its neighborhood, applying a movement over all vertices. Once the strategy carries out a complete evaluation of all neighbors, it selects the one with the best improvement with respect to the current solution cost. Then, the strategy utilizes the selected solution to execute the next search step. Due to this way of exploring the neighborhood, its computing time is expected to be considerably higher than the FI strategy.

Finally, we propose a hybrid approach which combines FI and BI traverse methods, the Hybrid Improvement (HI). Given a solution to improve, the strategy starts selecting a vertex randomly. It utilizes the FI method, which will move such vertex across the neighborhood until it finds the first improving solution. However, instead of considering to pass this solution to the next search step, HI employs the BI method to evaluate all neighbors and select the best position for the vertex under evaluation in each step. Therefore, the HI combines the FI strategy with random selection to diversify the search in conjunction with the BI strategy to select in each step the positions that produce more benefit considering the objective function value. Thus, such a combination brings together the best of both FI and BI strategies. On the one hand, it is expected to be fast as FI since it does not have to evaluate the entire neighborhood in each step. On the other hand, it is supposed to explore better quality solutions than FI due to the fact that it analyses the whole search space for a given vertex as BI does.

3.4. Shake

The shake method is designed in VNS methodology to diversify the search inside a local search process. Its primary aim is to enable the local search to systematically change the neighborhood structure by applying a different movement of such local search. Consequently, this movement helps the search process to escape from local optimum and explore more promising regions of the search space. In particular, in this work, we have defined the shake method by considering the moves defined in Section 3.2. Input parameters are the incumbent solution, φ, and the size of the maximum perturbation, usually named as k_{max}. The method returns the corresponding perturbed solution, φ'. In order to favor the diversification, it uses a different move than the one used in the local search. Then, if the local search is configured with the insert move, the shake method uses the exchange move and vice versa. The proposed procedure works as it is customary in VNS designs, applying the corresponding move sequentially for k_{max} steps.

3.5. Neighborhood Change

The neighborhood change method is responsible for selecting the next neighborhood to be explored inside VNS. Algorithm 3 shows the pseudocode of the neighborhood change method proposed for the *2DBMP*.

Algorithm 3 *NeighborhoodChange*(φ, φ', k)

```
1: if $\mathrm{B}_{2D}(G, \varphi') < \mathrm{B}_{2D}(G, \varphi)$ then
2:     $\varphi \leftarrow \varphi'$
3:     $k \leftarrow 1$
4: else
5:     $k \leftarrow k + 1$
6: end if
```

It follows the classical neighborhood change approach of VNS: if the new solution φ' outperforms φ, an improvement has been found, and the search starts again from the first neighborhood. Otherwise, the search continues with the next predefined neighborhood, $k + 1$. It is worth mentioning that k is bounded by k_{max}.

4. Computational Results

This section reports a complete evaluation of the exact (CSP models) and heuristic (Basic VNS) strategies developed to solve the *2DBMP*. It has as main targets: to analyze the performance of each refinement implemented in the proposed constraint programming models; to tune the parameters of each metaheuristic and to evaluate the influence of each component; and finally, to determine the efficacy and efficiency of both approaches and, additionally, their limitations for solving the *2DBMP*. CSP models (i.e., M_1, M_2, and M_3) have been compiled with *MiniZinc* [38, 39], into *FlatZinc*[2] and then solved sequentially with *Gecode* 6.1.1, a built-in solver included within the *MiniZinc* software distribution. On the other hand, all the heuristics algorithms have been implemented in Java 11. All the experiments have been conducted on an Intel© Core™ i5-3470 CPU running at 3.20 GHz and with 8 Gb of RAM.

The benchmark instances used in the experiments presented in this section are divided into two subsets. The first one is composed of 45 topologically diverse host graphs[3] of small and medium size. This subset includes 15 *Cartesian products of graphs*, 3 *paths*, 3 *cycles*, 5 *wheels*, 6 *t-th powers of cycles* ($t \in \{2, 10\}$), 6 *bipartite graphs*, 2 *complete graphs*, 4 *r-level t-ary trees* and 1 *Petersen graph.* The order and size of these instances range in the intervals $5 \leq n \leq 21$ and $6 \leq m \leq 190$, respectively. The second subset contains 45 representative and diverse graphs taken from the Harwell–Boeing Sparse Matrix Collection[4], which were previously used in [40]. These instances have an order and size in the ranges $48 \leq n \leq 960$ and $78 \leq m \leq 7442$, respectively. As it was described in Section 1, each guest graph is embedded in a bi-dimensional grid with size $\lceil\sqrt{n}\rceil \times \lceil\sqrt{n}\rceil$, being n the number of vertices of the guest graph.

We have conducted two types of experiments. On the one hand, preliminary experiments (Sections 4.1 and 4.2), which are designed for selecting the best CSP variant and finding the best values for the parameters required for the BVNS. On the other hand, final experiments (Section 4.3) with the aim of evaluating the performance of the best CSP and BVNS variants. The preliminary experiments have been conducted over a group of 21 instances from the first subset to avoid overfitting, while the final experimentation considers all the 90 benchmark instances.

4.1. CSP Results

In the first set of experiments we evaluate the performance of the three proposed constraint programming models devised for the *2DBMP*. As it was described in Section 2, an instance is given by a model and some data. In our case, the model can be one of the M_i, with $1 \leq i \leq 3$, introduced in this paper. The data in our instances correspond to the adjacency information of a particular host graph, i.e., the array *graph* is "filled", along with the lower (*lb*) and upper (*ub*) bounds of each guest graph. It is given in a *.dzn* file. Hence, the same model can be employed with various host graphs.

Table 1 presents the computational results obtained in our comparisons. For each selected benchmark graph, this table lists the name of the guest graph, the number of vertices (n), the number of edges (m), the lower (*lb*), and upper (*ub*) bounds in the first five columns. In order to determine the effectiveness of the proposed CSP models, we also include the number of calls to the propagation functions (*Prop*), where each call indicates the reduction of the search space by removing values that cannot participate in any solution;

[2]A standard input language which is supported by a large number of constraint programming solvers.
[3]Available at `https://www.tamps.cinvestav.mx/~ertello/2dbmp.php`
[4]`http://math.nist.gov/MatrixMarket/data/Harwell-Boeing`

the number of nodes of the search tree that have been explored (*Nodes*), which represents the nodes of the search tree that have been reached after a tentative enumeration for a variable; the number of explored nodes that failed to lead to a solution (*Failures*), that accounts for the explored nodes with an empty search space; the optimal (*Opt*) bi-dimensional bandwidth cost found; and the computational time (T(s)), in seconds, expended to reach the corresponding solution. All this information is presented for each one of the three CSP models evaluated. Certain of the considered host graphs could not be solved within the maximum CPU time allowed (72 hours), therefore the corresponding cells are marked with the symbol "–".

It can be observed from Table 1 that none of the analyzed models was able to provide a solution (neither optimal, nor sub-optimal), within the prefixed maximum CPU time, for the instances *wheel20, cyclePow15-10, cyclePow20-10,* and *bipartite10-10.* In the particular case of the graph *k4k5*, the model M_1 found a solution with cost value 4 (marked with symbol *star*). However, the solver using this basic model was unable to prove the optimality of this solution when it reached the allotted computational time, even when this is in fact the optimal cost value found by the more refined models M_2 and M_3. For the remaining host graphs all the proposed models found optimal solutions. When comparing the average computational time (see last row in Table 1), expended by these models for the *2DBMP*, one can remark that the slowest model is M_1. It is followed by M_2, which is 9.363% faster than it, thanks to the use of the *all_different* global constraint. The less time consuming model is M_3, which significantly reduced the search space by complementing M_2 with symmetry breaking constraints. Indeed, M_3 decremented 18.028% the average computational time employed by M_1. The search space reduction attained by M_3 with respect to M_1 and M_2 can be easily corroborated by contrasting the average number of nodes explored (*Nodes*) in the search tree. M_3 examined in average 5.33E+07 nodes while M_1 and M_2 traversed 1.16E+08 and 1.43E+08 nodes, respectively. This represents a reduction of 53.907% and 62.783% attained by M_3 with respect to the other two models. We can also remark that M_3 explores less "unnecessary" nodes, i.e., failure nodes. While M_1 and M_2 explore respectively 5.78E+07 and 7.16E+07 failure nodes, M_3 reduce this number to 2.66E+07, and is thus more efficient.

The results of this experiment provide valuable information about the optimal solution cost found for the selected host graphs. This information could be used to evaluate the performance of metaheuristic algorithms specially designed for solving the *2DBMP*, as it is shown in subsequent experiments.

Table 1: Performance comparison of three different constraint programming models for the *2DBMP*.

					M_1					M_2					M_3				
Graph	n	m	lb	ub	*Prop*	*Nodes*	*Failures*	*Opt*	*T*	*Prop*	*Nodes*	*Failures*	*Opt*	*T*	*Prop*	*Nodes*	*Failures*	*Opt*	*T*
p2p3	6	7	1	3	2.08E+03	7.20E+01	3.40E+01	1	0.148	1.12E+03	5.00E+01	2.20E+01	1	0.178	4.35E+02	2.00E+01	7.00E+00	1	0.200
p3p3	9	12	1	4	6.97E+03	1.63E+02	7.90E+01	1	0.221	1.54E+03	6.00E+01	2.70E+01	1	0.156	1.63E+03	5.10E+01	2.30E+01	1	0.257
p4p5	20	31	1	6	1.04E+06	9.13E+03	4.56E+03	1	0.536	4.81E+05	9.37E+03	4.67E+03	1	0.328	5.15E+04	9.90E+02	4.83E+02	1	0.220
p2c3	6	9	2	3	9.41E+02	2.30E+01	1.00E+01	2	0.187	2.45E+02	9.00E+00	1.00E+00	2	0.150	2.45E+02	9.00E+00	1.00E+00	2	0.243
p3c3	9	15	2	4	5.72E+03	1.32E+02	6.30E+01	2	0.175	7.47E+02	2.60E+01	6.00E+00	2	0.190	7.47E+02	2.60E+01	6.00E+00	2	0.198
p4c5	20	35	2	6	3.38E+05	3.28E+03	1.64E+03	2	0.400	6.56E+05	1.09E+04	5.44E+03	2	0.446	7.91E+05	1.44E+04	7.20E+03	2	0.526
c3c3	9	18	2	4	1.46E+04	2.84E+02	1.40E+02	2	0.168	8.92E+02	2.10E+01	5.00E+00	2	0.183	8.92E+02	2.10E+01	5.00E+00	2	0.236
c3c4	12	24	2	4	2.79E+04	4.65E+02	2.29E+02	2	0.179	2.77E+04	5.23E+02	2.56E+02	2	0.186	3.10E+03	5.80E+01	2.30E+01	2	0.283
c4c5	20	40	2	6	1.27E+05	1.22E+03	6.04E+02	2	0.309	1.98E+05	2.38E+03	1.18E+03	2	0.285	1.86E+04	2.35E+02	1.05E+02	2	0.278
k3k4	12	30	2	4	1.41E+07	1.64E+05	8.21E+04	3	3.744	6.29E+06	9.25E+04	4.62E+04	3	2.684	1.13E+06	1.69E+04	8.44E+03	3	0.548
k4k5	20	70	3	6	–	–	–	$\star$4	–	6.34E+11	4.71E+09	2.36E+09	4	176992.241	2.46E+11	1.83E+09	9.17E+08	4	74192.708
c3k4	12	30	2	4	1.40E+07	1.62E+05	8.12E+04	3	3.509	6.08E+06	9.20E+04	4.60E+04	3	1.823	1.10E+06	1.61E+04	8.05E+03	3	0.604
c4k5	20	60	2	6	4.32E+07	2.87E+05	1.44E+05	3	11.546	1.21E+07	1.64E+05	8.18E+04	3	4.553	5.41E+06	7.04E+04	3.52E+04	3	2.181
p3k4	12	26	2	4	3.12E+05	4.47E+03	2.23E+03	2	0.296	3.55E+04	5.60E+02	2.73E+02	2	0.190	3.26E+03	4.60E+01	1.60E+01	2	0.271
p4k5	20	55	2	6	2.52E+07	1.81E+05	9.04E+04	3	6.887	9.21E+06	1.35E+05	6.74E+04	3	3.317	4.95E+06	6.91E+04	3.46E+04	3	1.930
path10	10	9	1	4	1.40E+04	3.92E+02	1.91E+02	1	0.248	5.53E+02	2.80E+01	6.00E+00	1	0.393	6.43E+02	2.90E+01	6.00E+00	1	0.303
path15	15	14	1	5	2.19E+04	3.39E+02	1.66E+02	1	0.323	3.70E+03	1.27E+02	5.40E+01	1	0.306	1.82E+03	6.60E+01	2.20E+01	1	0.351
path20	20	19	1	6	3.66E+05	4.68E+03	2.33E+03	1	0.426	3.91E+03	1.44E+02	5.60E+01	1	0.360	3.08E+03	1.09E+02	3.90E+01	1	0.305
cycle10	10	10	1	4	1.26E+04	3.50E+02	1.70E+02	1	0.342	6.24E+02	2.90E+01	6.00E+00	1	0.350	6.70E+02	2.90E+01	6.00E+00	1	0.242
cycle15	15	15	1	5	6.87E+06	9.34E+04	4.67E+04	2	2.083	2.07E+06	8.11E+04	4.06E+04	2	1.464	4.31E+05	1.59E+04	7.93E+03	2	0.502
cycle20	20	20	1	6	3.68E+05	4.73E+03	2.36E+03	1	0.516	1.26E+04	5.62E+02	2.68E+02	1	0.344	1.02E+04	4.34E+02	2.04E+02	1	0.277
wheel5	5	8	1	2	3.48E+03	1.33E+02	6.50E+01	2	0.232	1.98E+03	9.00E+01	4.30E+01	2	0.257	1.46E+03	5.70E+01	2.80E+01	2	0.209
wheel7	7	12	2	3	1.95E+04	4.41E+02	2.20E+02	2	0.242	9.07E+03	2.63E+02	1.31E+02	2	0.346	4.86E+03	1.34E+02	6.50E+01	2	0.234
wheel10	10	18	2	4	1.92E+06	3.54E+04	1.77E+04	2	0.774	2.49E+05	5.91E+03	2.95E+03	2	0.415	1.37E+05	3.04E+03	1.52E+03	2	0.408
wheel15	15	28	3	5	2.25E+11	3.93E+09	1.97E+09	3	52574.381	4.52E+10	1.00E+09	5.02E+08	3	16196.897	1.29E+10	2.89E+08	1.45E+08	3	4444.857

Continued on next page ...

Table 1 – Continued from previous page

					M_1					M_2					M_3				
Graph	n	m	lb	ub	$Prop$	$Nodes$	$Failures$	Opt	T	$Prop$	$Nodes$	$Failures$	Opt	T	$Prop$	$Nodes$	$Failures$	Opt	T
wheel20	20	38	3	6	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
cyclePow10-2	10	20	2	4	2.42E+04	4.49E+02	2.22E+02	2	0.313	7.75E+03	1.42E+02	6.60E+01	2	0.432	4.71E+03	8.20E+01	3.60E+01	2	0.251
cyclePow15-2	15	30	2	5	7.75E+04	1.11E+03	5.48E+02	2	0.295	1.66E+05	3.18E+03	1.58E+03	2	0.513	8.64E+04	1.59E+03	7.88E+02	2	0.279
cyclePow20-2	20	40	2	6	1.49E+05	1.47E+03	7.29E+02	2	0.381	2.56E+04	3.89E+02	1.77E+02	2	0.412	3.99E+03	6.60E+01	1.70E+01	2	0.296
cyclePow10-10	10	45	4	4	4.64E+03	4.80E+01	2.20E+01	4	0.316	1.30E+03	1.90E+01	3.00E+00	4	0.360	1.30E+03	1.90E+01	3.00E+00	4	0.206
cyclePow15-10	15	105	5	5	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
cyclePow20-10	20	190	6	6	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
bipartite3-3	6	9	2	3	8.64E+02	2.00E+01	9.00E+00	2	0.365	6.70E+02	2.00E+01	9.00E+00	2	0.473	6.70E+02	2.00E+01	9.00E+00	2	0.256
bipartite3-4	7	12	2	3	5.92E+05	1.37E+04	6.85E+03	3	0.502	4.04E+05	9.97E+03	4.98E+03	3	0.618	1.93E+05	5.11E+03	2.55E+03	3	0.331
bipartite4-4	8	16	2	3	6.23E+05	1.26E+04	6.29E+03	3	0.505	5.83E+05	1.09E+04	5.43E+03	3	0.715	1.81E+05	4.25E+03	2.13E+03	3	0.333
bipartite5-5	10	25	2	4	3.18E+06	4.33E+04	2.17E+04	3	1.156	2.97E+06	4.46E+04	2.23E+04	3	1.494	1.42E+06	2.13E+04	1.06E+04	3	0.777
bipartite7-8	15	56	3	5	2.52E+10	2.68E+08	1.34E+08	4	7254.319	1.44E+10	1.13E+08	5.66E+07	4	4331.583	3.51E+09	2.99E+07	1.49E+07	4	1050.305
bipartite10-10	20	100	3	6	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
petersen	10	15	2	4	1.73E+04	3.71E+02	1.82E+02	2	0.302	1.70E+04	4.48E+02	2.17E+02	2	0.297	1.16E+03	3.80E+01	1.00E+01	2	0.310
complete5	5	10	1	2	1.64E+04	4.13E+02	2.06E+02	2	0.258	8.28E+03	2.33E+02	1.16E+02	2	0.389	7.77E+03	2.15E+02	1.07E+02	2	0.280
complete10	10	45	2	4	3.32E+08	3.34E+06	1.67E+06	4	72.457	2.39E+08	2.41E+06	1.20E+06	4	72.373	5.17E+07	5.48E+05	2.74E+05	4	14.025
tree2-2	7	6	1	3	2.16E+03	7.80E+01	3.70E+01	1	0.235	1.38E+03	8.60E+01	3.90E+01	1	0.473	2.63E+03	1.27E+02	6.10E+01	1	0.309
tree2-3	13	12	1	4	8.39E+05	1.75E+04	8.74E+03	2	0.513	1.40E+05	6.34E+03	3.17E+03	2	0.502	1.56E+05	5.93E+03	2.96E+03	2	0.307
tree3-2	15	14	1	5	1.48E+06	2.35E+04	1.17E+04	2	0.646	2.79E+05	9.82E+03	4.90E+03	2	0.545	1.66E+04	5.70E+02	2.76E+02	2	0.247
tree2-4	21	20	2	6	2.41E+10	4.18E+08	2.09E+08	2	6805.339	1.44E+09	4.04E+07	2.02E+07	2	735.881	1.14E+09	3.17E+07	1.59E+07	2	546.790
Average					6.86E+09	1.16E+08	5.78E+07		30283.244	1.70E+10	1.43E+08	7.16E+07		27447.895	6.44E+09	5.33E+07	2.66E+07		24823.640

4.2. Algorithm parameter tuning

This section is devoted to the selection of the best parameters of the proposed algorithm. We have selected a group of 21 representative instances from the first subset of benchmarks for this preliminary experiment in order to avoid overfitting. We report in this experiment the following metrics: *Avg*, the average bi-dimensional bandwidth obtained by the considered algorithm; T(s), the average computing time in seconds; *Dev*(%), the average deviation with respect to the best solution found in the experiment; and #*Best*, the number of times that the algorithm matches the best solution of the corresponding experiment.

The first experiment is devoted to select the best value for the α parameter of the constructive procedures. As mentioned in Section 3.1, the smaller the value, the more greedy the algorithm is, and vice versa, so it is recommended to test different values in the range 0-1. In particular, we have selected $\alpha = \{0.25, 0.50, 0.75\}$ to evaluate a very greedy variant, a balanced one, and a rather random variant, respectively. In order to test the robustness of the proposed approach, each procedure is configured to construct 100 solutions over each instance. Notice that each method returns the best solution found among the 100 constructed ones and the computing time per instance is computed as the sum of time to construct those 100 solutions. This values per instance are finally averaged over the 21 instances and reported in Table 2.

	α	*Avg.*	T(s)	*Dev*(%)	#*Best*
	0.25	4.10	0.22	3.97	19
C1	0.50	4.05	0.11	2.38	20
	0.75	4.05	0.09	1.59	20
	0.25	4.29	0.19	16.27	16
C2	0.50	4.19	0.09	13.49	18
	0.75	4.19	0.09	10.32	17

Table 2: Effect of the α parameter in each constructive procedure.

The results show that the computing time is negligible for the constructive procedure, being always smaller than 1 second. Regarding the quality of the solutions obtained, it can be clearly seen that *C1* consistently produces better results than *C2*, reaching a deviation of 1.59% in its best variant ($\alpha = 0.75$), while *C2* is not able to go below 10%. Additionally, *C1* is able to reach 20 out of 21 best solutions, while the best variant of *C2* matches the best solution 18 times. Notice that introducing randomness in both constructive procedures lead us to reach better solutions. Analyzing these results, we select *C1* with $\alpha = 0.75$ for the remaining experiments.

In the next experiment we test the performance of the three local search strategies described in Section 3.3 (First, Best, and Hybrid Improvements) when exploring the two introduced neighborhoods in Section

3.2 ($N_{Ins}(\cdot)$ and $N_{Exc}(\cdot)$). Table 3 shows the associated results for the corresponding 6 local search variants, reporting the same metrics than above. In order to isolate the contribution of each local search method, all of them start from the solutions constructed with the same algorithm (i.e., *C1* with $\alpha = 0.75$). As in the aforementioned experiment, each algorithm is executed 100 iterations over each instance, returning the best solution found and the sum of computing times in executing 100 iterations. Then, the results are averaged over the subset of 21 instances used in the preliminary experimentation.

		Avg.	T(s)	*Dev*(%)	*#Best*
N_{Ins}	LS_{FI}	3.95	206.82	33.75	7
	LS_{BI}	3.80	263.34	30.83	9
	LS_{HI}	3.80	355.77	26.42	7
N_{Exc}	LS_{FI}	3.50	21.30	20.42	11
	LS_{BI}	3.25	30.69	11.67	16
	LS_{HI}	3.25	41.85	11.67	16

Table 3: Effect of each Local Search in the $N_{Ins}(\cdot)$ and $N_{Exc}(\cdot)$ neighborhoods.

As we can see in this table, local search strategies based on the exploration of $N_{Exc}(\cdot)$ obtains consistently better results than those based on $N_{Ins}(\cdot)$. This fact can be partially explained by the nature of the *2DBMP* problem. Specifically, there exists some problems where the objective function is more sensitive to the absolute positioning of the elements in the solution than to their relative positioning. The *2DBMP* problem falls into the first category, where interchange moves usually drive to better outcomes. In particular, an interchange move produces a new solution where only two elements have changed their corresponding positions (see Figure 3). On the other hand, insert moves modify the position of much more elements in the resulting solution (see Figure 4).

Among all variants, LS_{BI} and LS_{HI} (based on the exploration of $N_{Exc}(\cdot)$) emerge as the best ones in terms of all the considered metrics. Therefore, we embed both strategies in a Basic Variable Neighborhood Search scheme, the first one, denoted as BVNS1, configured with $C1(0.75)$, LS_{BI}, and $N_{Exc}(\cdot)$; while the other one, denoted as BVNS2, is composed of $C1(0.75)$, LS_{HI}, and $N_{Exc}(\cdot)$. We then conduct a new experiment to select the best value for the $k_{\max}$ parameter for BVNS, which indicates the largest neighborhood to be explored. Figure 5 shows the average deviation obtained when considering $k_{\max} = \{0.1 \cdot n, 0.2 \cdot n, 0.3 \cdot n, 0.4 \cdot n, 0.5 \cdot n\}$. It is important to remark that $k_{\max}$ value is dependant of the size of the instance, facilitating the scalability of the algorithm. To complement this information, we additionally include in this figure the associated computing time in seconds (number close to each graph point).

These results are in line with the basis of VNS methodology [30], which indicates that the value of

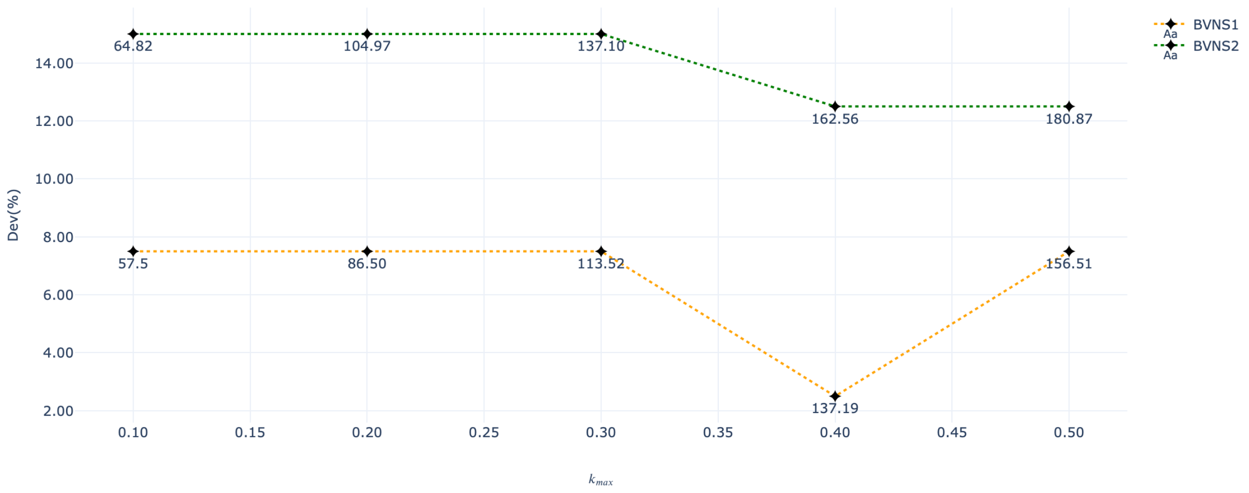


Figure 5: Comparison of different $k_{\max}$ values for the configured two BVNS strategies.

$k_{\max}$ should not be large. The rationale behind this is that a large perturbation will result in a complete different solution, converting VNS in a multi-start algorithm, which is not the main aim of the framework. Symmetrically, $k_{\max}$ should not be too small, since the search procedure could be trapped in the same basin of attraction.

The best results in both variants are obtained when considering $k_{\max} = 0.4 \cdot n$. As expected, the larger the $k_{\max}$, the larger the CPU time. It is worth mentioning that these two BVNS variants present better results than the multi-start procedure (100 iterations of construction coupled with local search) reported in Table 3. Specifically, the objective function values of BNVS1 and BVNS2 averaged over the 21 instances used in the preliminary experimentation are 2.50% and 12.50%, respectively.

Comparing the results obtained with each method, it seems that BVNS1 experimentally performs better than BVNS2. Specifically, the former obtains better results in terms of average deviation and computing time than the latter. Therefore, we select BVNS1 as our best algorithm. For the sake of simplicity, we refer this method to as *BVNS* in the remaining experiments.

4.3. Final experiments

This section is devoted to compare the best configuration of *BVNS* with the CSP models (Section 2) by considering the whole subset of 45 small and medium size instances. The main objective is to evaluate whether VNS is able to find optimal values (certified with CSP) or not. Table 4 shows a summary of the results of CSP and VNS. We report: *Avg*, the average bi-dimensional bandwidth obtained with each method; T(s), the average computing time in seconds; Dev(%), the average deviation with respect to the best solution found in this experiment; #*Opt*, the number of times that the algorithm matches the optimal value; and, #*Best*, the number of times that the algorithm matches the best solution of the corresponding experiment. As in previous experiments each method returns the best solution found among the 100 independent iterations, and the computing time per instance is computed as the sum of time to generate those 100 solutions. This values per instance are finally averaged over the 45 small and medium size instances and reported in Table 4.

	Avg.	*T(s)*	*Dev(%)*	*#Opt*	*#Best*
M_1	2.49	1362745.98	2.30	40	42
M_2	2.49	1235155.29	2.30	41	42
M_3	2.49	1117063.78	2.30	41	42
BVNS	2.55	321.66	11.11	35	39

Table 4: Comparison between BVNS and CSP models over the 45 small and medium size instances.

As expected, *BVNS* is extremely fast when comparing it with the three variants of CSP. It finds the optimal value in 35 instances (out of 45) in few seconds (7.15 on average). Additionally, in the four instances where none of the CSP variants certify the optimality, *BVNS* finds equal or even better results than the corresponding upper bound of M_1, M_2, and M_3. In those instances where CSP variants obtain better results, the deviation achieved with VNS seems to be large. It is worth mentioning that the optimal value of most instances is really small (close to 1). Consequently, if the optimal value of an instance is 1 and *BVNS* finds a solution with objective function value of 2, it will result in a 100% of deviation. Therefore, 11.1% can be considered as a good result, even more considering the average objective function value (2.49 vs 2.55), which indicates that, in those instances where *BVNS* is not able to reach the optimal value, it remains very close to it.

It is important to remark that the performance of the three CSP models for small and medium size instances is really competitive since they find the optimal solutions for a considerable number of instances in very short computing time. We then conduct a complementary experiment to analyze the scalability of M_3 and *BVNS* when solving larger instances. In order to do so, we have employed our second subset of instances taken from the Harwell–Boeing Sparse Matrix Collection.

Table 5 shows the individual results per instance of each considered procedure. As in Table 1, columns 1 to 3 show the name of the guest graph, number of vertices (n), and number of edges (m). Then, for M_3 and *BVNS*, we report: the best bi-dimensional bandwidth cost ($O.F.$); the average gap with respect to the best-known value (*BestKnown* is either the optimum or the lower bound), computed as

$$\%Gap = \frac{Val - BestKnown}{BestKnown} \cdot 100$$

where Val is the objective function value obtained with either M_3 or $BVNS$; and the computational time in seconds (T(s)). The cases where the instance is not solved within the maximum CPU time allowed (72 hours) are marked with the symbol ($\star$). As in the aforementioned experiments, $BVNS$ is executed 100 iterations over each instance, returning the best solution found, and the sum of computing times in executing 100 iterations. Then, the results are averaged over the subset of 45 Harwell–Boeing instances.

As we can see in this table, M_3 finds better solutions than $BVNS$ in only two instances. Specifically, M_3 matches the optimal value in *can_62* in less than 10 seconds ($BVNS$ finds a solution one unit above the optimal cost). In *nos6*, M_3 finds a feasible solution with value 5 in 72h, while $BVNS$ reaches a solution with value 12 in 960 seconds. In 27 instances, M_3 finds feasible solutions, but it was unable to certify the optimality. In those instances, $BVNS$ reaches considerably better solutions in shorter computing times. Finally, in 16 instances, M_3 was not able to even improve in 72h the initial upper bound. On the contrary, $BVNS$ considerably improves the upper bound, obtaining results close to the lower bound in moderate computing times.

To further investigate the performance of the proposed procedure, we conduct a convergence analysis by considering time-to-target plots (TTTPlot), which is essentially a run-time distribution. It creates a time-to-target solution value plots for measured CPU times that are assumed to be a shifted exponential distribution. This is often the case in local search-based heuristics for combinatorial optimization. See [41, 42, 43, 44, 45, 46] for another alternatives of studying the convergence analysis of algorithms. Figure 6 shows the time-to-target plots of four representative instances.

The experimental hypothesis in this time-to-target plots is that running times fit a two parameter exponential distribution. Given an instance, we store the execution time needed to find an objective function value at least as good as a given target value. In this case, the algorithm is executed a certain number of times on the selected instance and using the given target solution. The random number generator is initialized with a different seed in each run and, therefore, the executions are assumed to be independent. To compare both empirical and theoretical distributions, we follow a standard graphical methodology for data analysis [47], executing our proposal 30 times, and recording for each run the running time required to match the target value. The abscissa axis represents running time, while the ordinate axis reports the probability of obtaining the best-known value. The resulting graph confirms the expected exponential run-time distribution of our algorithm.

4.4. Managerials implications

The method developed in this work can find high-quality solutions reducing the time drastically if we compare to the exact algorithm proposed. Thus, the developed constructives offer procedures to build feasible solutions which could have high confidence in reaching a high-quality solution that could be corresponding to an optimal solution. Conversely, if non-well-quality solutions are built, different methods to traverse

			M3			*BVNS*		
Graph	*n*	*m*	*O. F.*	*%Gap*	*T(s)*	*O. F.*	%Gap	*T(s)*
bcsstk01	48	176	6	1.00	259200.01	5	0.67	48.00
can___62	62	78	**2**	1.00	**9.86**	3	2.00	62.00
nos4	100	247	8	3.00	259200.01	5	1.50	100.01
bcspwr03	118	179	($\star$)15	6.50	259200.09	4	1.00	118.01
bcsstk04	132	1758	($\star$)16	3.00	259200.11	10	1.50	132.01
bcsstk22	138	279	8	7.00	259200.07	4	3.00	138.01
can__144	144	576	($\star$)16	7.00	259200.07	5	1.50	144.01
bcsstk05	153	1135	($\star$)17	7.50	259200.01	7	2.50	153.01
can__161	161	608	9	3.50	259200.08	6	2.00	161.01
dwt__198	198	597	10	4.00	259200.07	3	0.50	198.01
dwt__209	209	767	10	4.00	259200.02	6	2.00	209.01
dwt__221	221	704	10	9.00	259200.09	6	5.00	221.01
can__229	229	774	($\star$)21	9.50	259200.09	5	1.50	229.01
dwt__234	234	300	($\star$)21	9.50	259200.07	2	0	234.01
nos1	237	390	10	9.00	259200.05	4	3.00	237.01
dwt__245	245	608	11	4.50	259200.04	6	2.00	245.02
lshp_265	265	744	11	10.00	259200.08	6	5.00	265.00
bcspwr04	274	669	12	5.00	259200.07	6	2.00	274.02
ash292	292	958	12	11.00	259200.06	6	5.00	292.00
can__292	292	1124	($\star$)24	7.00	259200.08	7	1.33	292.00
dwt__307	307	1108	12	5.00	259200.01	6	2.00	307.00
dwt__310	310	1069	12	11.00	259200.09	7	6.00	310.00
dwt__361	361	1296	13	12.00	259200.09	7	6.00	361.01
plat362	362	2712	13	5.50	259200.01	8	3.00	362.01
bcsstk07	420	3720	($\star$)28	13.00	259200.09	9	3.50	420.01
bcspwr05	443	590	15	6.50	259200.01	6	2.00	443.01
can__445	445	1682	15	4.00	259200.01	8	1.67	445.01
bcsstk20	485	1325	15	14.00	259200.09	6	5.00	485.01
494_bus	494	586	16	7.00	259200.04	6	2.00	494.01
dwt__503	503	2762	($\star$)31	14.50	259200.01	9	3.50	503.01
lshp_577	577	1656	16	15.00	259200.07	8	7.00	577.00
dwt__607	607	2262	($\star$)34	10.33	259200.09	4	0.33	607.00
662_bus	662	906	($\star$)36	17.00	259200.05	6	2.00	662.01
nos6	675	1290	*5*	4.00	259200.05	12	11.00	960.01
685_bus	685	1282	18	8.00	259200.06	7	2.50	685.01
can__715	715	2975	($\star$)37	11.33	259200.09	9	2.00	715.01
nos7	729	1944	19	8.50	259200.01	9	3.50	729.01
dwt__758	758	2618	($\star$)38	37.00	259200.02	7	6.00	758.01
lshp_778	778	2247	19	18.00	259200.01	9	8.00	778.01
bcsstk19	817	3018	($\star$)40	39.00	259200.01	8	7.00	817.00
dwt__878	878	3285	20	9.00	259200.09	9	3.50	878.00
gr_30_30	900	3422	21	9.50	259200.01	9	3.50	900.01
dwt__918	918	3233	($\star$)42	41.00	259200.08	9	8.00	918.01
nos2	957	1590	21	20.00	259200.06	6	5.00	957.01
nos3	960	7442	($\star$)43	20.50	259200.01	12	5.00	960.01
	Average		12.50	4.73	253440.27	6.71	3.38	439.63

Table 5: Results obtained by *BVNS* in large instances where CSP model M_3 finds feasible solutions, but it was unable to certify their optimality after 72 hours.

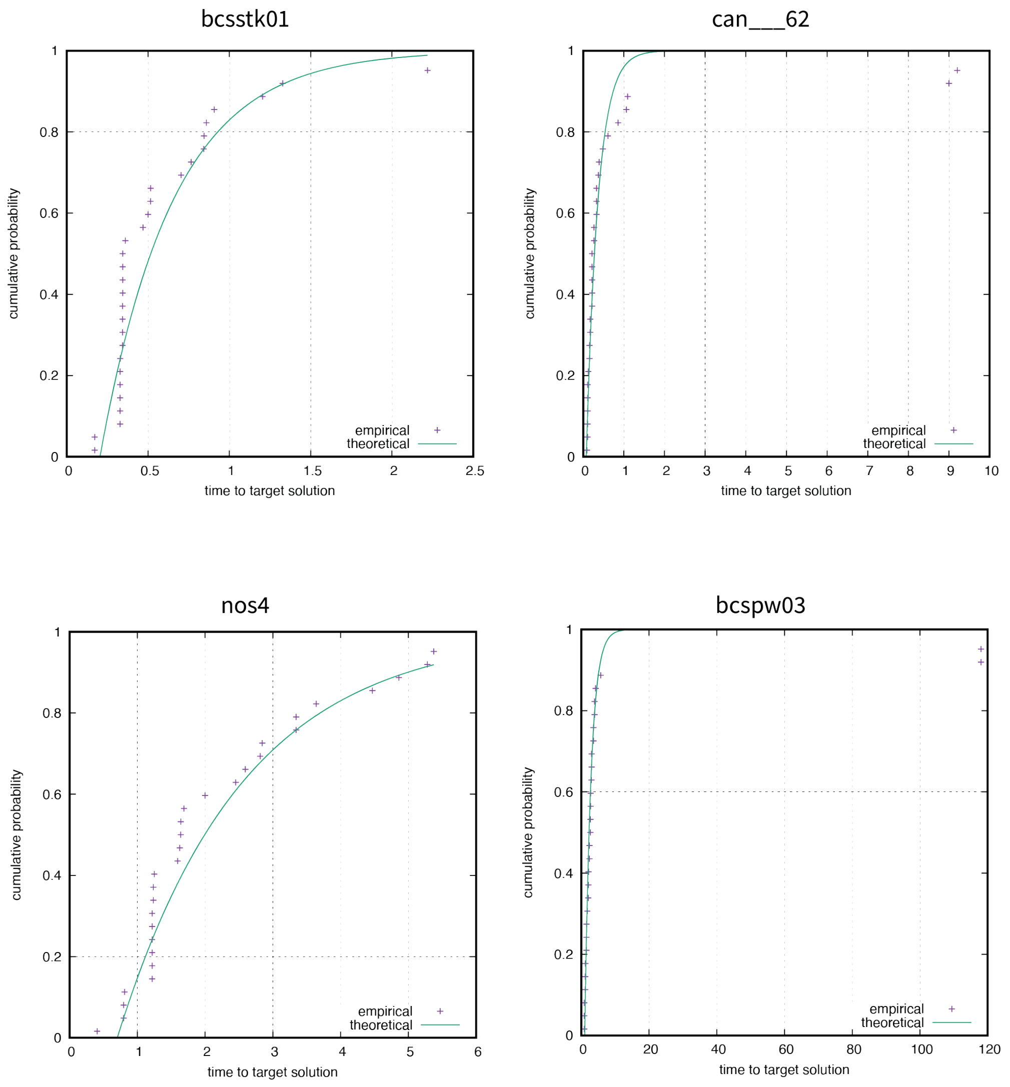


Figure 6: Time-to-target plots for 4 representative instances.

the space search are analyzed, which enable us to reach better solutions, even the instances where the contructives fails. Finally, for those complicated instances, a trajectorial change is applied when a local optimum is achieved by using BVNS strategy. The procedures and metrics designed could be directly applicable on, for instance, VLSI design.

5. Conclusions

This work tackles the two-dimensional bandwidth reduction from both exact and heuristic perspectives. On the one hand, several Constraint Satisfaction Problem models are proposed with the aim of finding the optimal solution for small instances. This algorithm becomes unpractical when dealing with large instances. On the other hand, a metaheuristic algorithm is proposed, following the Variable Neighborhood Search framework. The computational results show the efficacy and efficiency of both approaches and, additionally, their limitations. Our best CSP model finds the optimal solution in small instances in few seconds but, for large instances it reaches low-quality values in 72 hours. The BVNS is able to reach the optimal value for most of the instances where the optimal value is known in very short computing times. Additionally, the method is able to provide high quality solutions for those instances in which the CSP model is not able to reach the optimal value in reasonable computing times. We do believe that our experimental findings, algorithms, and benchmark instances can be useful for the scientific community as a framework to compare both, new exact and heuristic proposals.

We have identified different avenues to continue this research. Specifically, we do believe that there is still room to improve the current heuristic and exact approaches for this problem. Additionally, the study of the links between the *2DBMP* and VLSI design may help to develop more efficient algorithms. In this case, circuits are usually modeled as multi-graphs (different types of components and/or connections) that should be mapped onto a regular structure. Therefore we might need to define a different objective function (multiple edges between a pair of nodes, weights in either nodes or edges, etc.). This modification in the objective function is able to guide the algorithm to explore an alternative search space, which will eventually lead to an improvement of the solution quality. In this line, it would be also interesting to explore in the near future the mapping to more complex structures such as 3D grids, trees, etc.

Acknowledgments

This work has been partially founded by Research Talent Attraction Program by the Comunidad de Madrid with grants references 2017-T2/TIC-5664, P2018/TCS-4566 and Young Researchers R+D Project. Ref. M2173 – SGTRS (co-funded by Rey Juan Carlos University), respectively. The third author's work was partially supported by the Mexican Secretariat of Public Education under Grant SEP-Cinvestav (2019-2020) No. 00114.

References

[1] S. L. Bezrukov, J. D. Chavez, L. H. Harper, M. Röttger, U. P. Schroeder, Embedding of hypercubes into grids, in: L. Brim, J. Gruska, J. Zlatuška (Eds.), Mathematical Foundations of Computer Science 1998, Springer Berlin Heidelberg, Berlin, Heidelberg, 1998, pp. 693–701. doi:doi.org/10.1007/BFb0055820.

[2] S. N. Bhatt, F. T. Leighton, A framework for solving VLSI graph layout problems, Journal of Computer and System Sciences 28 (2) (1984) 300–343. doi:10.1016/0022-0000(84)90071-0.

[3] J. Opatrny, D. Sotteau, Embeddings of complete binary trees into grids and extended grids with total vertex-congestion 1, Discrete Applied Mathematics 98 (3) (2000) 237–254. doi:10.1016/S0166-218X(99)00161-4.

[4] C. H. Papadimitriou, The NP-completeness of the bandwidth minimization problem, Computing 16 (3) (1976) 263–270. doi:10.1007/BF02280884.

[5] V. Chvátal, A remark on a problem of harary, Czechoslovak Mathematical Journal 20 (1) (1970) 109–111.

[6] E. M. Gurari, I. H. Sudborough, Improved dynamic programming algorithms for bandwidth minimization and the mincut linear arrangement problem, Journal of Algorithms 5 (4) (1984) 531–546. doi:10.1016/0196-6774(84)90006-3.

[7] G. M. Del Corso, G. Manzini, Finding exact solutions to the bandwidth minimization problem, Computing 62 (3) (1999) 189–203. doi:10.1007/s006070050002.

[8] A. Caprara, J.-J. Salazar-González, Laying out sparse graphs with provably minimum bandwidth, INFORMS Journal on Computing 17 (3) (2005) 356–373. doi:10.1287/ijoc.1040.0083.

[9] R. Martí, V. Campos, E. Piñana, A branch and bound algorithm for the matrix bandwidth minimization, European Journal of Operational Research 186 (2) (2008) 513–528. doi:10.1016/j.ejor.2007.02.004.

[10] E. Cuthill, J. McKee, Reducing the bandwidth of sparse symmetric matrices, in: Proceedings of the 1969 24th ACM national conference, Association for Computing Machinery, 1969, pp. 157–172. doi:10.1145/800195.805928.

[11] N. E. Gibbs, W. G. Poole, Jr, P. K. Stockmeyer, An algorithm for reducing the bandwidth and profile of a sparse matrix, SIAM Journal on Numerical Analysis 13 (2) (1976) 236–250. doi:10.1137/0713023.

[12] R. Martí, M. Laguna, F. Glover, V. Campos, Reducing the bandwidth of a sparse matrix with tabu search, European Journal of Operational Research 135 (2) (2001) 450–459. doi:10.1016/S0377-2217(00)00325-8.

[13] E. Pinana, I. Plana, V. Campos, R. Martí, Grasp and path relinking for the matrix bandwidth minimization, European Journal of Operational Research 153 (1) (2004) 200–210. doi:10.1016/S0377-2217(02)00715-4.

[14] A. Lim, J. Lin, B. Rodrigues, F. Xiao, Ant colony optimization with hill climbing for the bandwidth minimization problem, Applied Soft Computing 6 (2) (2006) 180–188. doi:10.1016/j.asoc.2005.01.001.

[15] G. Czibula, G.-C. Crişan, C.-M. Pintea, I.-G. Czibula, Soft computing approaches on the bandwidth problem, Informatica 24 (2) (2013) 169–180. doi:10.15388/Informatica.2013.390.

[16] E. Rodriguez-Tello, J.-K. Hao, J. Torres-Jimenez, An improved simulated annealing algorithm for bandwidth minimization, European Journal of Operational Research 185 (3) (2008) 1319–1335. doi:10.1016/j.ejor.2005.12.052.

[17] N. Mladenovic, D. Urosevic, D. Pérez-Brito, C. G. García-González, Variable neighbourhood search for bandwidth reduction, European Journal of Operational Research 200 (1) (2010) 14–27. doi:10.1016/j.ejor.2008.12.015.

[18] J. Torres-Jimenez, I. Izquierdo-Marquez, A. Garcia-Robledo, A. Gonzalez-Gomez, J. Bernal, R. N. Kacker, A dual representation simulated annealing algorithm for the bandwidth minimization problem on graphs, Information Sciences 303 (2015) 33–49. doi:10.1016/j.ins.2014.12.041.

[19] F. R. K. Chung, Labelings of graphs, in: L. W. Beineke, R. J. Wilson (Eds.), Selected topics in graph theory volume 3, Academic Press, 1988, Ch. 7, pp. 151–168.

[20] L. Lin, Y. Lin, Two models of two-dimensional bandwidth problems, Information Processing Letters 110 (11) (2010) 469 – 473. doi:10.1016/j.ipl.2010.04.013.

[21] S. N. Bhatt, S. S. Cosmadakis, The complexity of minimizing wire lengths in VLSI layouts, Information Processing Letters 25 (4) (1987) 263 – 267. doi:10.1016/0020-0190(87)90173-6.

[22] Z. Miller, J. Orlin, NP-completeness for minimizing maximum edge length in grid embeddings, Journal of Algorithms 6 (1) (1985) 10 – 16. doi:10.1016/0196-6774(85)90016-1.

[23] L. Lin, Y. Lin, Square-root rule of two-dimensional bandwidth problem, RAIRO - Theoretical Informatics and Applications 45 (4) (2011) 399–411. doi:10.1051/ita/2011120.

[24] Y. Lin, On density lower bounds of two dimensional bandwidth, Journal of Mathematical Research and Exposition 16 (3) (1996) 343–349.

[25] A. D. Gordon, T. A. Henzinger, A. V. Nori, S. K. Rajamani, Probabilistic programming, in: Future of Software Engineering Proceedings, FOSE 2014, Association for Computing Machinery, New York, NY, USA, 2014, p. 167–181. doi:10.1145/2593882.2593900.

[26] F. Rossi, P. van Beek, T. Walsh (Eds.), Handbook of Constraint Programming, Vol. 2 of Foundations of Artificial Intelligence, Elsevier, 2006.

[27] J. Jaffar, J. Lassez, Constraint logic programming, in: Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages (POPL'87), Munich, Germany, January 21-23, 1987, 1987, pp. 111–119.

[28] J. Puget, On the satisfiability of symmetrical constrained satisfaction problems, in: Methodologies for Intelligent Systems, 7th International Symposium, ISMIS '93, Trondheim, Norway, June 15-18, 1993, Proceedings, 1993, pp. 350–361.

[29] J. M. Crawford, M. L. Ginsberg, E. M. Luks, A. Roy, Symmetry-breaking predicates for search problems, in: Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96), Cambridge, Massachusetts, USA, November 5-8, 1996, 1996, pp. 148–159.

[30] P. Hansen, N. Mladenović, J. A. M. Pérez, Variable neighbourhood search: methods and applications, Annals of Operations Research 175 (1) (2010) 367–407. doi:10.1007/s10479-009-0657-6.

[31] A. Duarte, J. Sánchez-Oro, M. G. Resende, F. Glover, R. Martí, Greedy randomized adaptive search procedure with exterior path relinking for differential dispersion minimization, Information Sciences 296 (2015) 46–60. doi:10.1016/j.ins.2014.10.010.

[32] J. Sánchez-Oro, A. Duarte, S. Salcedo-Sanz, Robust total energy demand estimation with a hybrid variable neighborhood search – extreme learning machine algorithm, Energy Conversion and Management 123 (2016) 445–452. doi:10.1016/j.enconman.2016.06.050.

[33] T. A. Feo, M. G. Resende, S. H. Smith, A greedy randomized adaptive search procedure for maximum independent set, Operations Research 42 (5) (1994) 860–878.

[34] J. Sánchez-Oro, M. Laguna, A. Duarte, R. Martí, Scatter search for the profile minimization problem, Networks 65 (1) (2015) 10–21. doi:10.1002/net.21571.

[35] J. Sánchez-Oro, J. J. Pantrigo, A. Duarte, Combining intensification and diversification strategies in VNS. an application to the vertex separation problem, Computers & Operations Research 52 (2014) 209–219. doi:10.1016/j.cor.2013.11.008.

[36] A. J. McAllister, A new heuristic algorithm for the linear arrangement problem, Tech. Rep. TR-99-126a, Faculty of Computer Science, University of New Brunswick (1999).

[37] J. J. Pantrigo, R. Martí, A. Duarte, E. G. Pardo, Scatter search for the cutwidth minimization problem, Annals of Operations Research 199 (1) (2012) 285–304. doi:10.1007/s10479-011-0907-2.

[38] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, G. Tack, MiniZinc: Towards a Standard CP Modelling Language, in: C. Bessière (Ed.), Principles and Practice of Constraint Programming – CP 2007, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 529–543. doi:10.1007/978-3-540-74970-7_38.

[39] P. J. Stuckey, T. Feydy, A. Schutt, G. Tack, J. Fischer, The MiniZinc challenge 2008–2013, AI Magazine 35 (2) (2014) 55–60. doi:10.1609/aimag.v35i2.2539.

[40] A. Duarte, L. F. Escudero, R. Martí, N. Mladenovic, J. J. Pantrigo, J. Sánchez-Oro, Variable neighborhood search for the vertex separation problem, Computers & Operations Research 39 (12) (2012) 3247–3255. doi:10.1016/j.cor.2012.04.017.

[41] A. Gharaei, S. A. Hoseini Shekarabi, M. Karimi, Modelling and optimal lot-sizing of the replenishments in constrained, multi-product and bi-objective epq models with defective products: Generalised cross decomposition, International Journal of Systems Science: Operations & Logistics 7 (3) (2020) 262–274.

[42] A. Gharaei, M. Karimi, S. A. Hoseini Shekarabi, Joint economic lot-sizing in multi-product multi-level integrated supply chains: Generalized benders decomposition, International Journal of Systems Science: Operations & Logistics (2019) 1–17.

[43] S. A. Hoseini Shekarabi, A. Gharaei, M. Karimi, Modelling and optimal lot-sizing of integrated multi-level multi-wholesaler supply chains under the shortage and limited warehouse space: generalised outer approximation, International Journal of Systems Science: Operations & Logistics 6 (3) (2019) 237–257.

[44] C. Duan, C. Deng, A. Gharaei, J. Wu, B. Wang, Selective maintenance scheduling under stochastic maintenance quality with multiple maintenance actions, International Journal of Production Research 56 (23) (2018) 7160–7178.

[45] A. Gharaei, M. Karimi, S. A. H. Shekarabi, An integrated multi-product, multi-buyer supply chain under penalty, green, and quality control polices and a vendor managed inventory with consignment stock agreement: The outer approximation with equality relaxation and augmented penalty algorithm, Applied Mathematical Modelling 69 (2019) 223–254.

[46] A. Gharaei, S. A. Hoseini Shekarabi, M. Karimi, E. Pourjavad, A. Amjadian, An integrated stochastic epq model under quality and green policies: generalised cross decomposition under the separability approach, International Journal of Systems Science: Operations & Logistics (2019) 1–13.

[47] J. M. Chambers, Graphical methods for data analysis, CRC Press, 2018.

[48] J.-X. Hao, Two-dimensional Bandwidth of Mobius Ladders and Other Graphs, Henan Science 18 (1) (2000) 15–20.

Appendix A. Graphs with known bounds

1. For any graph G with n vertices and diameter $D(G)$ [24],

$$\frac{\delta(n)}{D(G)} \leq \mathrm{B}_{2D}(G) \leq \delta(n) \ ,$$

where

$$\delta(n) = \min\left\{2\left\lceil\frac{\sqrt{2n-1}-1}{2}\right\rceil,\ 2\left\lceil\sqrt{\frac{n}{2}}\right\rceil - 1\right\} \ .$$

2. *k-level complete binary trees.* Such a tree, denoted $T_{2,k,}$, has $n = 2^k$ vertices and its two-dimensional bandwidth value has the upper bound [24]:

$$\mathrm{B}_{2D}(T_{2,k,}) \geq \left\lceil \frac{1}{k-1} \left\lceil \frac{\sqrt{2^{k+1}-3}-1}{2} \right\rceil \right\rceil \ .$$

3. *Rook graphs.* These graphs are constructed as the Cartesian product two complete graphs K_{n_1} and K_{n_2}.

$$\left\lceil \frac{n-1}{2} \right\rceil \leq \mathrm{B}_{2D}(K_{n_1} \times K_{n_2}) \leq n-1 \ .$$

and the bounds are tight [23].

Appendix B. Graphs with known optimal solution

1. *Complete graphs.* A complete graph K_n is a simple undirected graph with n vertices in which every pair of distinct vertices is connected by an edge. The optimal two-dimensional bandwidth for a complete graph K_n is:

$$\mathrm{B}_{2D}(K_n) = \delta(n) \ ,$$

the reader is referred to [24] for details.

2. *Stars.* A star graph S_n is a tree of order n constructed with one central vertex and $n-1$ leaves. Thus, a star S_n is isomorphic to the complete bipartite graph $K_{1,n-1}$. According to [24], the optimal two-dimensional bandwidth for a star S_n is:

$$\mathrm{B}_{2D}(S_n) = \left\lceil \frac{\delta(n)}{2} \right\rceil \ .$$

3. *Complete bipartite graphs.* A complete bipartite graph K_{n_1,n_2} is a graph whose vertex set is decomposed into two independent sets $n_1 = |V_1|$ and $n_2 = |V_2|$ (*i.e.*, no two vertices within the same set are adjacent) such that every pair of vertices, $v_1 \in V_1$ and $v_2 \in V_2$, are adjacent. In [48] it was established that for $n_1 \leq n_2$,

$$\mathrm{B}_{2D}(K_{n_1,n_2}) = \left\lceil \frac{\delta(n_1) + \delta(n_1+n_2) - 1}{2} \right\rceil \ ,$$

if n_1, $n_1 + n_2$ are inadequate for $\delta(n_1)$, $\delta(n_1+n_2)$ respectively. Otherwise,

$$\mathrm{B}_{2D}(K_{n_1,n_2}) = \left\lceil \frac{\delta(n_1) + \delta(n_1+n_2)}{2} \right\rceil \ .$$

Where an integer n is said to be inadequate to $\delta(n)$ when $\delta(n) = 2r$, $n \leq r(2r+1)$; when $\delta(n) = 2r+1$, $n \leq (r+1)(2r+1)$

4. *Cycles.* A cycle graph C_n is build as a circular arrangement of n vertices such that all of them have degree two. If n is even,

$$\mathrm{B}_{2D}(C_n) = 1 \ ;$$

if n is odd,

$$\mathrm{B}_{2D}(C_n) = 2 \ .$$

5. *Combs.* The comb D_n is a caterpillar with n nonpendant vertices each incident with exactly one pendant vertex. According to [48] its optimal two-dimensional bandwidth cost is,

$$\mathrm{B}_{2D}(D_n) = 1 \ .$$

6. *Wheels.* A wheel W_n consists of an $(n-1)$-cycle (called the rim) every point of which is joined to a single common point (called the hub) by a line (called a spoke) [48].

$$\mathrm{B}_{2D}(D_n) = \max\left\{ \left\lceil \frac{\delta(n)}{2} \right\rceil, 2 \right\} \ .$$

7. *Möbius ladders.* The Möbius ladder M_n is a cubic circulant graph with an even number n of vertices, formed from an n-cycle by adding edges (called "rungs") connecting opposite pairs of vertices in the cycle [48].
$$\mathrm{B}_{2D}(M_n) = 2\ .$$
8. *Petersen.* For the Petersen graph G the optimal two-dimensional bandwidth value is [23]:
$$\mathrm{B}_{2D}(G) = 2\ .$$
9. *Two dimensional meshes.* These graphs are constructed as the Cartesian product of two paths P_{n_1} and P_{n_2} [23].
$$\mathrm{B}_{2D}(P_{n_1} \times P_{n_2}) = 1\ .$$
10. *Cylinder grids.* These graphs are constructed as the Cartesian product of a path P_{n_1} and a cycle C_{n_2} ($n_1 \geq 2$, $n_2 \geq 3$) [23].
$$\mathrm{B}_{2D}(P_{n_1} \times C_{n_2}) = 2\ .$$
11. *Torus grids.* These graphs are constructed as the Cartesian product two cycles C_{n_1} and C_{n_2} ($n_1, n_2 \geq 3$) [23].
$$\mathrm{B}_{2D}(C_{n_1} \times C_{n_2}) = 2\ .$$
12. *Triangulated triangles.* For the triangulated triangles T_n, the two-dimensional bandwidth is [23]:
$$\mathrm{B}_{2D}(T_n) = 2\ .$$