Heuristics for the Min-Max Arc Crossing Problem in Graphs

Rafael Martí*
Universitat de València, Spain. Rafael.Marti@uv.es

Vicente Campos
Universitat de València, Spain. Vicente.Campos@uv.es

Arild Hoff
Molde University College, Norway. Arild.Hoff@hiMolde.no

Juanjo Peiró
Universitat de València, Spain. Juanjo.Peiro@uv.es

Abstract

In this paper we study the visualization of complex structures in the context of automatic graph drawing. Constructing geometric representations of combinatorial structures, such as networks or graphs, is a difficult task that requires an expert system. The automatic generation of drawings of graphs finds many applications from software engineering to social media. The objective of graph drawing expert systems is to generate layouts that are easy to read and understand. This main objective is achieved by solving several optimization problems. In this paper we focus on the most important one: reducing the number of arc crossings in the graph. This hard optimization problem has been studied extensively in the last decade, proposing many exact and heuristic methods to minimize the total number of arc crossings. However, despite its practical significance, the min-max variant in which the maximum number of crossings over all edges is minimized, has received very little attention. We propose new heuristic methods based on the strategic oscillation methodology to solve this NP-hard optimization problem. Our experimentation shows that the new method compares favorably with the existing ones, implemented in current graph drawing expert systems. Therefore, a direct application of our findings will improve these functionality (i.e., crossing reduction) of drawing systems.

KeyWords: Graph drawing expert system, metaheuristics, crossing reduction, iterated greedy.

Original Version: September 6th, 2017
Revised version: February 1st, 2018.

**** Corresponding author**: Rafael Martí. Universitat de València, Facultat de Ciències Matemàtiques. Doctor Moliner, 50. 46100 – Burjassot (Valencia), Spain.*

1. Introduction

Graphs are nowadays a modeling tool to represent and analyze data in many areas from production to business reengineering. The term graph drawing refers to the problem of constructing geometric representations of graphs, where drawing conventions depend upon the application and context in which the graph is used, and where arc crossing minimization is probably the most important aesthetic criteria considered. As stated by Carpano (1980) in a seminal paper in the graph drawing field, "the most crucial problem as far as readability of a graph is that of arc crossing".

In the last years, many areas in science, business and engineering have experienced an enormous growth in terms of the amount of data that they analyze. As a matter of fact, the term Big Data was recently coined to reflect this phenomenon. In this context, the representation of large graphs, and in particular the development of graph drawing expert systems, has received increasing interest. The first interactive system, called Grab (Rowe et al., 1987), was basically a graph editor and it is usually referred to as a first-generation system. Tom Sawyer software company (www.tomsawyer.com) created GraphEd (Himsolt, 1996), which can be considered the first expert system in this context. It implements layout algorithms for automatic graph drawing.

Nowadays we can find different expert systems tailored for special types of graphs. Jünger and Mutzel (2004) describe 14 software systems based on several standards. These authors identified in their text different types of representations, such as circular, orthogonal (grid), clustered, compound, and layered or hierarchical. Each application domain typically employs one of these types. The book by Di Battista et al. (1999) is a reference in the area of graph drawing and explains in detail these standards and their associated ascetic criteria to obtain a readable layout. For example, in project management, activity networks are usually represented as layered digraphs where vertices are constrained to lie on a set of equally spaced horizontal or vertical lines, and edges flow in the same direction, as shown in Figure 1. In this paper we focus on layered drawings.

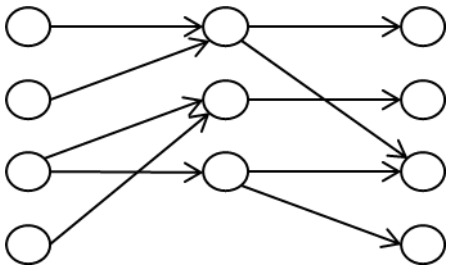

Figure 1. Layered digraph.

The so-called Sugiyama's method (1981) to represent digraphs according to the layered standard, has led to several drawing expert systems. This method first assigns vertices to layers, adding dummy vertices to model long edges. This first step is called *Layer Assignment*. Then, in the second step, the method orders the vertices in each layer, usually with the barycenter algorithm, for *Arc Crossing Minimization* (Martí and Laguna, 2003). Finally, in the third step called *Coordinate Assignment*, it allocates the vertices in a specific position in their layer to reduce arc length and bends of long edges. In this way, any digraph can be represented as a proper

hierarchical or layered graph. This makes this graphic convention (i.e., hierarchical graphs) a popular standard in the field. In this paper we consider the second step of this graph drawing system: the optimization problem consisting in the minimization of the number of arc crossings in a layered graph. This is a difficult problem (it is NP-hard), and constitutes a challenge for optimization methods.

If we restrict our attention to the layered representation of graphs (also called hierarchical), we can identify several graph drawing expert systems that implement algorithms to obtain it. Table 1 summarizes the most popular ones in our opinion, specifying how they reduce arc crossings.

Drawing System	**Characteristic**	**Crossing reduction method**
Graphviz pygraphviz.github.io/	Hierarchical static graphs	Median heuristic coupled with local search (exchanges)
Dynagraph www.dynagraph.org	Dynamic graphs	Median heuristic (adapted to incremental drawing)
yEd www.yworks.com	Editor with layout methods for different representations.	Barycenter and median heuristics
MSAGL www.microsoft.com	Microsoft Graph Layout. Constrained to given space	Ordering rules
AGD www.ads.tuwien.ac.at	Library of algorithms for several classes of layouts	Multi-start barycenter from random orderings.
Tom Sawyer www.tomsawyer.com	Software development kit for various layout styles	Ordering rules
Mathematica www.wolfram.com/mathematica/	Wolfram Language for the aesthetic drawing of graphs.	LayeredGraphPlot ranks vertices for arc crossing minimization

Table 1. Graph Drawing Expert Systems for Layered Graphs

The problem of arc crossing minimization in hierarchical graphs has been extensively studied. First efforts, such as Eades and Kelly (1986), restricted themselves to simple ordering rules and graphs with only two layers. More elaborated procedures, based on metaheuristic methodologies, such as tabu search (Laguna et al., 1997) or GRASP (Laguna and Martí, 1999), were later introduced to obtain improved outcomes. A computational comparison of 16 procedures on 900 randomly generated bipartite graphs was presented by Martí and Laguna (2003). This study shows that the procedures based on modern metaheuristics dominate those

based on ordering rules in terms of solution quality but at the expense of more computational time. Early developments sacrificed solution quality in favor of speed, considering the latter a critical factor in automated drawing systems. This is why, graph drawing expert systems, as those shown in Table 1, implement simple heuristic procedures, such as the median or the barycenter, to solve this difficult problem. So, in terms of crossing minimization, we can say that these expert systems provide a very fast solution of medium quality. The contribution of this paper is therefore to propose an efficient algorithm for crossing reduction to improve this functionality of expert drawing systems.

Figure 2 shows a small size graph (100 vertices in 5 layers and 255 arcs) with an arbitrary node ordering (i.e., without minimizing arc crossings). It clearly illustrates how hard can be to analyze a “non-optimized” graph drawing.

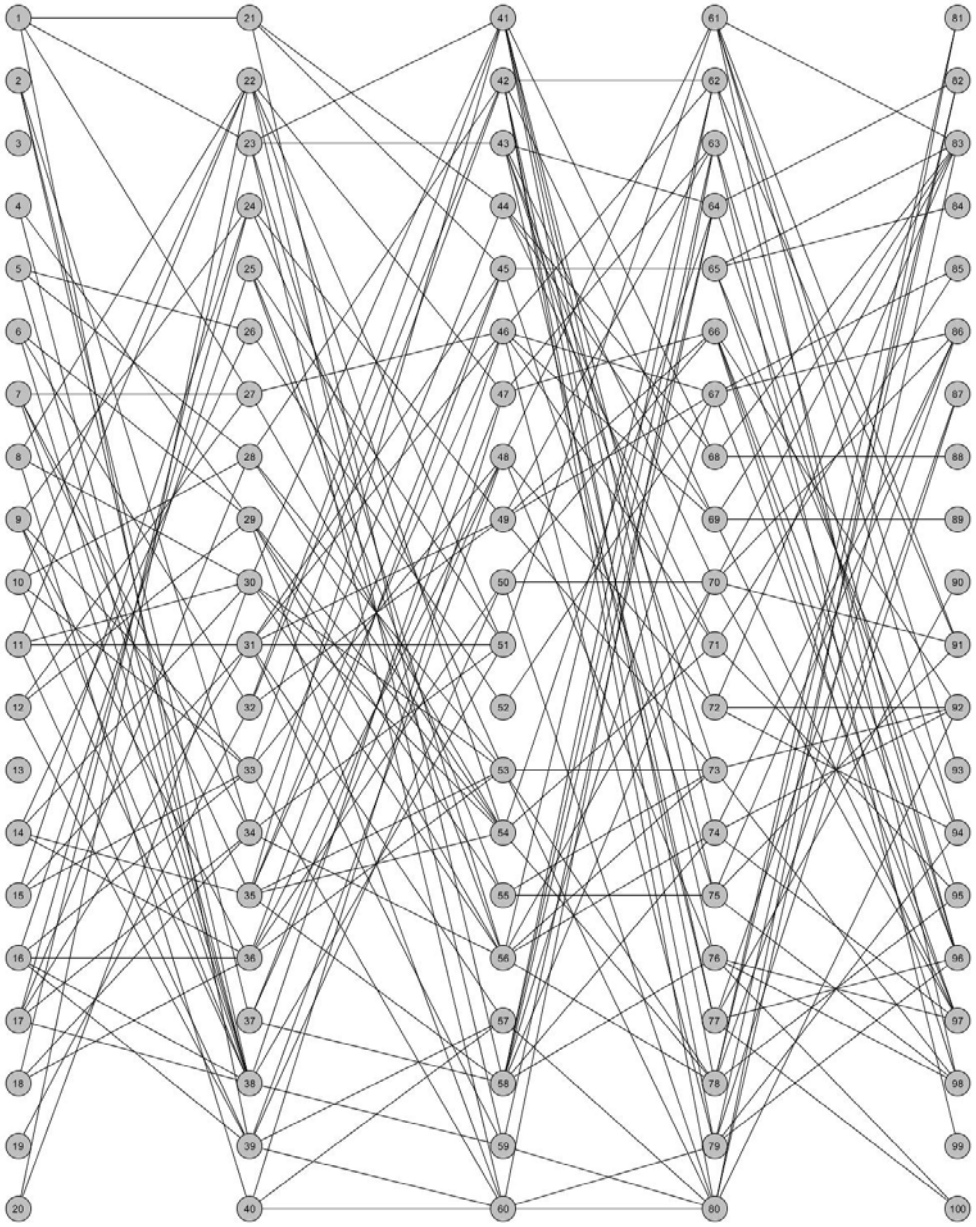

Figure 2. An arbitrary drawing of a graph with 100 nodes and 255 arcs.

All the methods described above minimize the total number of arc crossings in a graph. There is however a recent method devoted to minimize the maximum number of arc crossings among all the edges of the graph. Stallmann (2012) identified some applications (i.e. Bhatt and Leighton, 1984) in the context of VLSI circuits in which it is more appropriate to minimize the maximum number of crossings over all edges (min-max problem) than minimizing the traditional sum of crossings (min-sum problem). He calls this variant the *bottleneck crossing problem* and, in line with his proposal, we have empirically found that solutions to the min-max problem usually result in more readable graphs than solutions to the min-sum problem. This is especially evident in graph drawing tools where zooming highlights a specific area of the graph and the overall crossing reduction does not imply a low number of crossings in the zoomed area. Stallmann proposed the maximum crossing edge (MCE) heuristic specifically designed for the bottleneck problem, but his experimentation shows that this heuristic also obtains competitive solutions with respect to the sum of crossings. In other words, this author considers the min-max as the primary objective function, and the min-sum as the secondary one, reporting both values in the experimentation to assess the merit of the MCE heuristic.

The main contributions of this work are:

1. Proposing, implementing and testing a new heuristic for crossing minimization
2. Proposing, implementing and testing a new mathematical model
3. Experimental comparison of the two new solving methods with an existing algorithm
4. Graphical comparison of the new solution method with graph drawing expert systems
5. Improvement of the state of the art in arc crossing minimization

Our new heuristic algorithm, based on the strategic oscillation (SO) methodology (Glover and Laguna, 1997), minimizes the maximum number of crossings over the edges of a graph and, as a subsidiary goal, the total (sum) number of crossings. As mentioned, we focus on hierarchical directed acyclic graphs (HDAG) which are also known as layered graphs. Note that working with HDAGs is not a limitation since there exists a number of procedures to transform a directed acyclic graph (DAG) into a HDAG (Sugiyama et al., 1981).

In the next section, we first introduce some notation and definitions, and in Section 3, we describe the previous MCE heuristic. Section 4 is devoted to the description of our strategic oscillation method for the min-max problem, which also considers the min-sum as a secondary objective. The experimentation in Section 5 shows that our method is able to compete with the previous method in both objectives. We apply statistical analysis to draw significant conclusions to finish the paper.

2. Notation and Formulation

A hierarchical graph $H = (V, E, k, L)$ is defined as a graph $G = (V, E)$, where V and E represent the set of vertices and edges, respectively, and the function $L(v)$: $V \rightarrow \{1,2, \dots, k\}$ indicates the index of the layer where v resides. The literature in graph drawing usually does not distinguish between the terms edge and arc, so we will use both to refer to the links in the graphs. The L function implicitly defines the sets of vertices $L_i = \{ v \in V : L(v) = i \}$ for $i = 1,2, \dots, k$ which we refer to as layers. Since the edges in a HDAG are straight lines that join the vertices in two contiguous layers, a *drawing* of a HDAG is given by the ordering of the vertices in each layer. In

mathematical terms, a drawing of H is defined as $D = (H, \Phi)$, where $\Phi = \{\varphi_1, \varphi_2, \ldots, \varphi_k\}$ and φ_i is the ordering (permutation) of the vertices in layer L_i. That is, $\varphi_i(j)$ is the vertex in position j in layer L_i. The position of vertex v in layer L_i is defined as $\pi_i(v)$ in such a way that if $v = \varphi_i(j)$ then $\pi_i(v) = j$ and conversely.

Let $c(e)$ be the crossing number of edge $e = (u, v)$ that represents the number of edges that cross edge $e \in E$. An arc crossing is produced between edges (u, v) and (u', v'), where $u, u' \in L_i$ and $v, v' \in L_{i+1}$ when:

$$\big(\pi(u) < \pi(u') \wedge \pi(v) > \pi(v')\big) \vee \big(\pi(u) > \pi(u') \wedge \pi(v) < \pi(v')\big).$$

The *maximum crossing number* of a drawing $D = (H, \Phi)$, that we denote by $mc(D)$, is the maximum of $c(e)$ across all the edges in H, i.e.:

$$mc(D) = max_{e \in E}\, c(e).$$

The *min-max arc crossing problem* in a HDAG may be formulated as the problem of finding the ordering in each layer in such a way that $mc(D)$ is minimum. An optimal drawing $D^\star$ is such that no other D has a lower value of $mc(D)$. In Stallmann (2012), the min-max arc crossing problem is called the *bottleneck crossing problem*.

The *crossing number* of a drawing D, $c(D)$, is the total (sum) number of crossings in the graph, i.e.:

$$c(D) = \frac{1}{2}\sum_{e \in E} c(e),$$

and the classical edge crossing minimization problem, called here the min-sum problem, consists of finding the ordering in each layer to minimize $c(D)$. Both combinatorial optimization problems, minimizing $mc(D)$ and minimizing $c(D)$, are NP-hard[1] (Garey and Johnson, 1983).

The maximum crossing edge heuristic (MCE) proposed in Stallmann (2012) is specifically designed to minimize $mc(D)$ as described below. Additionally, this method also optimizes $c(D)$ as a secondary objective.

Jünger and Mutzel (1997) proposed a linear model based on binary variables for the min-sum problem. For the sake of simplicity, we describe here the formulation for a 2-layered graph, which can be easily generalized to an arbitrary layered graph. In this formulation, variable $x_{ik} = 1$ when node i precedes node k in the first layer (called the left layer in 2-layered graphs). Similarly, $y_{lj} = 1$ when node l precedes node j in the second layer (called the right layer). The variable definition is completed with the introduction of c_{ijkl}, which takes the value 1 when a crossing between edges (i, j) and (k, l) occurs. To adapt this formulation to the min-max problem, we added variable $c(i, j)$ which indicates the number of crossing of edge (i, j); in other words, what we defined as the crossing number of the edge. With this notation, we formulate the problem as:

[1] Stallmann (2012) states that the proof by Garey and Johnson for the min-sum can be adapted to the min-max problem by using the bandwidth problem in place of the linear arrangement problem.

$$Min\ M$$

Subject to:

$$x_{ik} + y_{lj} - c_{ijkl} \leq 1 \quad (i,j),(k,l) \in E, i < k, j \neq l \quad (1)$$

$$x_{ki} + y_{jl} - c_{ijkl} \leq 1 \quad (i,j),(k,l) \in E, i < k, j \neq l \quad (2)$$

$$x_{ij} + x_{jk} + x_{ki} \leq 2 \quad 1 \leq i < j < k \leq n_1 \quad (3)$$

$$y_{ij} + y_{jk} + y_{ki} \leq 2 \quad 1 \leq i < j < k \leq n_2 \quad (4)$$

$$c(i,j) = \sum_{(k,l)\in E} c_{ijkl} + c_{klij} \quad (i,j) \in E \quad (5)$$

$$c(i,j) \leq M \quad (i,j) \in E \quad (6)$$

$$x_{ij} + x_{ji} = 1 \quad 1 \leq i < j \leq n_1 \quad (7)$$

$$y_{ij} + y_{ji} = 1 \quad 1 \leq i < j \leq n_2 \quad (8)$$

$$x_{ij}, y_{ij}, c_{ijkl} \in \{0,1\}$$

$$c(i,j), M \in \mathbb{Z}^+.$$

In this model, constraints (1) and (2) force c_{ijkl} to take the value 1 when the variables x_{ik} and y_{lj} indicate a crossing. Constraints (3) and (4) are the 3-dicycle constraints, which together with (7) and (8) guarantee that the ordering variables (x and y respectively) represent in fact an ordering. We included the new constraints (5) to compute the crossing number of all arcs and constraints (6) to force variable M to take the maximum of these crossing numbers, since the minimization of M in the objective function tries to reduce it as much as possible, thus providing the objective function value of the min-max problem. We will make use of this formulation in Section 5 to obtain the optimal results of some instances.

3. Previous methods

Traditionally, the 2-layer crossing minimization problem is solved by fixing the permutation φ_i with $i = 1, 2$ of the vertices in one of the two layers. Then, the permutation of vertices in the other layer is adjusted to best meet the objective of minimizing $c(D)$. This principle is also commonly used with multiple layers as the layers are considered one at a time and the position of the vertices is determined by looking at one of the adjacent layers. Bachmaier et al. (2010) and Matuszewski et al. (1999) describe the most popular of the classical heuristics: the barycenter, the median and the sifting algorithms. These heuristics are performed iteratively with three possible stopping criteria: either when no improvement is found in an iteration, after a given number of iterations or eventually after a pre-established amount of time.

Both the barycenter and the median algorithm use a *layer-by-layer sweep* technique where, first, an initial vertex ordering is determined for each layer $\{\varphi_1, \varphi_2, \dots, \varphi_k\}$. Then, starting at the second layer, the permutation φ_2 is decided with respect to the fixed ordering of φ_1. Using the barycenter, the position of a vertex v in layer L_2 is determined by calculating the mean position of vertices adjacent to v in layer L_1 and sorting the vertices in L_2 according to this number. This procedure is continued for each new layer L_i with respect to its previous layer L_{i-1} and when the last layer L_k is sorted, the procedure is repeated in decreasing order, where layer L_j considers the permutation in layer L_{j+1} instead. The median algorithm is similar to the barycenter with the exception that it uses the median instead of the mean of the adjacent vertices to sort the current layer.

The sifting algorithm is somewhat different, as it is based on vertex insertion instead of sorting. The vertices are chosen in a given order and inserted one by one sequentially. A chosen vertex v is inserted at the best position in its layer $L(v)$ with respect to the already included vertices. By using a given measure (initially $c(D)$), the optimal position is calculated and the vertex is inserted before the next one is handled in the same way. The degree of a vertex is defined as the number of edges incident to it, and the best strategy is considered to be the one starting with the vertex with highest degree and continues with the insertion of the following in a decreasing order. A *pass* is defined as the completion of the procedure of inserting all nodes once. After a pass, the sequence is reversed and each vertex is again inserted but now with respect to the current position of the other vertices.

Stallmann (2012) developed the MCE heuristic based on the sifting principle but designed to solve the bottleneck problem. He also showed that the MCE heuristic outperforms previous algorithms for solving the min-sum problem for certain instances, especially those with a large maximum vertex degree. MCE applies a different method than the classical sifting algorithm to find the optimal position for the vertices, although it also repeats passes until no further improvements can be found or another predefined stopping criterion is met.

The main principle of MCE is to use edges as basis for selecting vertices for insertion. Unlike the traditional sifting algorithm, which selects vertices according to the number of edges adjacent to them, MCE identifies the edge with the largest number of crossings and tries to reposition the vertices at the endpoints of this edge. Thus, the method starts with an initial drawing D, which can be determined by a complete random ordering of the vertices in each layer or with another pre-processing strategy. Then, it sorts the edges $e \in E$ in descending order according to their current number of crossings $c(e)$. At each step, following this order, an edge e is examined, and its endpoint vertices are checked in search for their best position. Note that instead of using the global $c(D)$ as the objective function to determine this best position, only the edges incident to the chosen vertex are considered. Thus, MCE determines the best position for a vertex v as the one that minimizes the maximum number of crossings among the edges incident to v. This means that a drawing could actually be poorer after a vertex insertion since the number of crossings of a non-adjacent edge could eventually increase. The author however explains that, as shown in the computational experiments, this strategy achieves good results, since it works as a diversification method by encouraging a variety of movements, thus preventing the search from being stuck in a local optimum. Stallmann (2012) also shows that using the barycenter for determining the initial drawing D, from which to apply MCE, can obtain

significantly better results. A post-processing procedure of exchanging adjacent vertices is also applied to further improve the solutions. The experimentation shows that, as expected, this method obtains a very low $mc(D)$ value, but additionally, it is also competitive with existing algorithms to minimize $c(D)$.

4. Strategic oscillation heuristic

Heuristic search procedures that aspire to find globally optimal solutions to hard combinatorial optimization problems usually require some type of diversification to overcome local optimality. A way to achieve diversification that has proven to be very effective (see, for example, Corberán et al., 2016) is to re-start the procedure from a new solution once a region has been explored. This is the core of the so-called multi-start procedures (Martí et al., 2013). These methods have become very popular in the last years, with probably the GRASP methodology (Festa and Resende, 2011) being one of the most applied when solving combinatorial optimization problems. Multi start methods usually alternate two phases, the first one in which a solution is built, and the second one in which this solution is usually improved by applying a local search procedure.

An interesting family within multi-start approaches is given by the rebuilding approaches in which, instead of building a solution from scratch at each iteration, the method employs some of the elements in the solution of the previous iteration to build a new one. As in multi-start methods, after generating a solution, a local search post-processing is usually applied. This rebuilding mechanism has revealed to be very effective to target hard optimization problems. In particular, the Strategic Oscillation methodology applies it to efficiently search the solution space. The generation of a solution by including certain elements from previous solutions follows basic tabu search principles (Glover and Laguna, 1997) since it is based on memory structures. In other words, it bases its exploration in recording certain information during the search process to find new efficient solutions. This contrasts with memory-less designs, such as GRASP, in which each iteration constitutes a new effort, not linked with the previous solutions.

In this paper we focus on a simplified version of the constructive and destructive processes, known as iterated greedy (IG) (Ruiz and Stützle, 2008). This method generates a sequence of solutions by iterating over a greedy constructive heuristic that, as in strategic oscillation, uses two main phases: destruction and construction. The IG method starts from a complete initial solution S and then iterates through a main loop. In this loop, it first generates a partial candidate solution S_p by removing a certain number of elements (destruction phase) from the complete candidate solution S and, next, reconstructs a complete solution starting from S_p (construction phase). Additionally, a local search phase is applied to the reconstructed solution in order to reach a local optimum. Before resorting to the next iteration in the main loop, an acceptance criterion decides whether the solution returned by the local search procedure becomes the new incumbent solution or not. The IG process alternates the destruction and construction phases with the local search post-processing until a termination criterion is met.

As mentioned, given a hierarchical graph $H = (V, E, k, L)$, a solution (or drawing) $D = (H, \Phi)$ provides an ordering of its vertices. Considering that the objective function of the min-max arc

crossing problem is given by $mc(D) = max_{e \in E}\, c(e)$, we define the critical edges, CE, to those with a crossing number equal to the objective function value. In mathematical terms:

$$CE = \{e \in E: c(e) = mc(D)\}$$

It is clear that if we want to improve a solution D with respect to $mc(D)$, we have to reduce the number of crossing of the critical edges. In this section, we propose both constructive and local search methods to reduce it. Considering that the computation of CE is time-consuming since it requires the exploration of all the edges in the graph, and that when we reduce the crossing of the edges in this set, other edges will become critical, we define the set of near-critical edges, $NCE(p)$, to deal simultaneously with both edges that are now critical, and edges that we expect to be critical in subsequent iterations of our algorithm. This set depends on a search parameter $p \in [0,1]$, measured as a percentage of the objective function value. In mathematical terms:

$$NCE(p) = \{e \in E: c(e) \geq p\, mc(D)\}.$$

Our constructive and local search phases have the primary objective of finding an ordering in the layers to reduce the number of crossings of the edges in $NCE(p)$, thus trying to improve the objective function value $mc(D)$. To do this, we define the set of near critical vertices, $NCV(p)$, as the set of vertices that are endpoints of the near critical edges, i.e.:

$$NCV(p) = \{v \in V: (u,v) \in NCE(p)\ or\ (v,u) \in NCE(p)\}.$$

The constructive mechanism applies the Iterated Greedy methodology, which as explained above, partially destroys a solution to obtain a new one from it. In the following subsections, we will describe how it operates. In short, given a solution, it basically removes the vertices in $NCV(p)$ plus a random selection, and repositions them according to the barycenter.

4.1 Constructive heuristic

Our constructive heuristic, outlined in Algorithm 1, starts by creating a candidate list CL of unassigned vertices, which at the beginning of the algorithm consists of all the vertices in the graph. The initial position of each vertex is assigned a value of zero ($\pi_i(v) = 0,\ \forall v \epsilon V$). The heuristic starts by randomly selecting a vertex from the vertices in CL with maximum degree. The vertex is placed in an arbitrary position (see steps 1 to 7 of Algorithm 1).

Once a selected vertex v has been positioned in the partial solution, CL is updated by deleting v. In subsequent construction steps (9 to 16 of Algorithm 1), the next vertex v is randomly selected from a restricted candidate list RCL that consists of vertices with a degree of no less than α $(0 \leq \alpha \leq 1)$ of the maximum degree d_{max} in CL. The vertex degree is calculated with respect to the subgraph given by the partial solution obtained from previous vertex selections. In mathematical terms $RCL = \{v \in CL: d(v, V \setminus CL) \geq \alpha\, d_{max}\}$, where $d(v, V \setminus CL)$ is the degree of vertex v with respect to the vertices already positioned (i.e., with respect to the partial solution under construction). In general terms, a selected vertex v is placed in its layer in the position prescribed by the barycenter, denoted by $bc(v)$, computed as the arithmetic mean of the positions of the already assigned adjacent vertices to v. The construction phase terminates after $|V|$ steps, when all vertices have been selected and positioned.

Input: (H, α)
1 Define $CL \coloneqq V$
2 **forall** $v \in CL$ **do**
3 $\pi_i(v) \coloneqq 0$
4 **end forall**
5 Randomly select $v^* \in CL$ according to a uniform distribution
6 Identify layer $i = L(v^*)$
7 Set an arbitrary position $\pi_i(v^*)$
8 Update $CL \coloneqq CL \setminus \{v^*\}$
9 **while** $CL \neq \emptyset$ **do**
10 Define $RCL = \{v \in CL : d(v, V \setminus CL) \geq \alpha d_{max}\}$
11 Randomly select $v^* \in RCL$ according to a uniform distribution
12 Identify layer $i = L(v^*)$
13 Compute barycenter $bc(v^*)$
14 Set position $\pi_i(v^*) \coloneqq bc(v^*)$
15 Update $CL \coloneqq CL \setminus \{v^*\}$
16 **end while**
Output: D

Algorithm 1. GRASP constructive phase of the Strategic Oscillation procedure.

The first iteration of the constructive heuristic operates as described above and outlined in Algorithm 1, which basically constitutes an implementation of a GRASP construction. In subsequent iterations, following the Strategic Oscillation methodology and the Iterated Greedy implementation, we feed the construction heuristic with a previous solution to keep the current position of some of its vertices, as shown in Algorithm 2.

Input: (D, p, β)
1 Define $NCE(p) = \{e \in E : c(e) \geq p \cdot mc(D)\}$
2 Define $NCV(p) = \{v \in V : (u, v) \in NCE(p) \text{ or } (v, u) \in NCE(p)\}$
3 **forall** $v \in NCV(p)$ **do**
4 Remove v from D
5 Set $\pi_i(v) \coloneqq 0$
6 **end forall**
7 $\text{counter} \coloneqq |NCV(p)|$
8 **while** $\text{counter} < \lfloor \beta |V| \rfloor$ **do**
9 Randomly select $v \in D \setminus NCV(p)$ according to a uniform distribution
10 Remove v from D
11 Set $\pi_i(v) \coloneqq 0$
12 $\text{counter} + +$
13 **end while**
14 define $CL = \{v \in V : \pi_i(v) = 0\}$
15 **while** $CL \neq \emptyset$ **do**
16 Define $RCL = \{v \in CL : d(v, V \setminus CL) \geq \alpha d_{max}\}$
17 Randomly select $v^* \in RCL$ according to a uniform distribution
18 Identify layer $i = L(v^*)$
19 Compute barycenter $bc(v^*)$
20 Set position $\pi_i(v^*) \coloneqq bc(v^*)$
21 Update $CL \coloneqq CL \setminus \{v^*\}$
22 **end while**
Output: D

Algorithm 2. Iterated Greedy phase of the Strategic Oscillation procedure.

Algorithm 2 shows that given a solution D, the Iterated Greedy phase of the Strategic Oscillation method starts by removing the associated set of near-critical vertices, $NCV(p)$ of D. Then, for diversification purposes, it removes if necessary an additional number of vertices randomly

selected from $D \setminus NCV(p)$ up to the target amount of $\beta|V|$. We consider that the position of all these "removed" vertices is assigned to a value of zero ($\pi_i(v) = 0$). Then, we re-apply the constructive method described above in Algorithm 1 to this partial solution by adding the removed vertices. This modified construction phase terminates when all the "removed" vertices have been selected and positioned, following the above-mentioned mechanisms, based on the use of RCL and the barycenter calculation. Once a solution D is completed, we evaluate it (compute the objective function value $mc(D)$), and submit it to the improvement heuristic.

Since we seek to optimize the $mc(D)$ value, we consider a modification of the barycenter calculation $bc(v)$ in the algorithms above. Instead of computing the average of the position of all its adjacent vertices, we compute in $bc(v)$ the weighted average of the positions of its near critical adjacent vertices (i.e., those in $NCV(p)$). Specifically, we consider the crossing number $c(v,u)$ as the weight to multiply the position of each near critical vertex u.

4.2 Improvement heuristic

Given a solution D, each step of the improvement heuristic, outlined in Algorithm 3, consists of selecting a vertex in the set of its near critical vertices $NCV(p)$ to be considered for a move. Since both the evaluation of a solution and the computation of this set is time consuming, we do not re-compute the solution value or the set after the application of a single move, but we directly resort to the next element in the set to explore its associated move. Once all the elements in $NCV(p)$ have been examined and eventually moved, we evaluate the resulting solution and compute its near critical vertex set for the next iteration of the improvement heuristic. One could argue that this implementation does not calculate the exact impact (or influence) of a move in the objective function at the time we perform it (the so-called move value). Although this is true, it is extremely fast in terms of computing time and can be indirectly controlled with the number of iterations without updating the solution value. This is done by adjusting the search parameter p, controlling the size of $NCV(p)$. This candidate list strategy, skipping the process of updating relevant information in each iteration of a method, has been successfully applied in the context of the tabu search methodology (Glover and Laguna, 1997).

When reviewing previous local search methods for edge crossing minimization (see Martí and Laguna, 2003), we found that some of them examine all the possible insertions for a vertex, while others limit themselves to a single position, usually the one given by the barycenter. The former methods require more computing time but usually find a better position than the latter ones, which only try one position in their moves. Here, we propose a mixed strategy to determine the position to insert a vertex in order to achieve a compromise between solution quality and speed.

When a vertex $v \in NCV(p)$ is selected (see step 7 in Algorithm 3), we compute its barycenter, $bc(v)$, with respect to the current ordering of its adjacent vertices. Then, the insertion move considers five positions: $bc(v)$, $bc(v)+1$, $bc(v)+2$, $bc(v)-1$, and $bc(v)-2$ (steps 10 to 15 in Algorithm 3). If some of these positions are not feasible (because of the size of the layer), we discard them. The vertex v is placed in the position, from these five, that produces the maximum reduction in the total (sum) number of crossings $c(D)$. If no reduction is possible, then the vertex is not moved. In any case, we resort to the next vertex in the near critical set. An improvement step terminates when all vertices have been considered for insertion (some

vertices may be moved while others may stay in their original positions.) Then, the final solution is evaluated (i.e. $mc(D)$ is computed), and the following step is performed from the new near critical set.

It must be noted that the selection of the vertices to be moved is computed based on the primary objective function $mc(D)$, and their insertion position is computed according to the secondary objective function $c(D)$. Our experimental analysis will confirm that the interplay of both evaluations is able to produce high quality solutions, in terms of both objectives, in short computational times.

Input: (D, p)
1 **do**
2 Compute $NCE(p) = \{e \in E : c(e) \geq p \cdot mc(D)\}$
3 Compute $NCV(p) = \{v \in V : (u, v) \in NCE(p) \text{ or } (v, u) \in NCE(p)\}$
4 Define $CL = NCV(p)$
5 **while** $CL \neq \emptyset$ **do**
6 Define $D' \coloneqq D$
7 Randomly select $v \in CL$ according to a uniform distribution
8 Identify layer $i = L(v)$
9 Set $\pi_i^*(v) \coloneqq \pi_i(v)$
10 **for** $pos = -2$ **to** $pos = +2$ **do**
11 In D', set position $\pi_i(v) \coloneqq bc(v) + pos$
12 **if** $\pi_i(v)$ is feasible **and** $c(D') < c(D)$ **then**
13 $\pi_i^*(v) \coloneqq \pi_i(v)$
14 **end if**
15 **end for**
16 In D', set position $\pi_i(v) \coloneqq \pi_i^*(v)$
17 Update $CL \coloneqq CL \setminus \{v\}$
18 **end while**
19 Set $improvement \coloneqq 0$
20 **if** $mc(D') < mc(D)$ **then**
21 $D \coloneqq D'$
22 $improvement \coloneqq 1$
23 **end if**
24 **while** $improvement == 1$
Output: D

Algorithm 3. Improvement method.

When Algorithm 3 finishes, we apply a post-processing to fine tune the solution and try to further reduce its $mc(D)$ value. In particular, we focus on the arc with maximum value (i.e. the critical arc responsible for the objective function value). If there is more than one critical arc, we chose one at random. Then, we scan all the possible positions for its end-nodes. We scan these positions one by one in search for the one with a maximum reduction in the $mc(D)$ value. To do so, we need to compute the crossing number of all the arcs in the layer. If we reduce $mc(D)$, we identify the new critical arc and continue in this way; otherwise we stop this post-processing. We only apply it selectively to promising solutions to reduce the total computational effort of our method. Specifically, we apply it to solutions with a $mc(D)$ value within 4 units of the best $mc(D)$ found so far.

5. Computational Experiments

As we mentioned in Section 3, to the best of our knowledge, Stallmann (2012) is the only previous work on the min-max arc crossing problem, so the primary purpose of our experimentation is to compare our method with his MCE heuristic. However, before performing this competitive testing, we undertake to explore the elements of our method in a scientific testing. In this way, we can calibrate and adjust the search parameters and evaluate its influence on the performance of the method. We employ 85 instances in our experimentation.

To follow-up on Stallmann's work, we employ his generator of instances in our experimentation. As he states, it is a challenge to obtain problem instances with varying degrees and possible "shapes". This author discards directed graphs as a source to obtain hierarchical graphs, since it is difficult to control the degree when applying standard layering methods, which usually add many "dummy" nodes. He considered a uniform random graph generator, where instances are denoted as $u(l, k, d, b)$ where l is the number of layers, k the number of nodes per layer, d is the graph density (in terms of the number of edges per vertex), and b is the bias in the random number generator. We generate 20 small instances (10 with $n = 60, l = 3$, and 10 with $n = 100, l = 5$), 30 medium instances ($n = 300, l = 15$ and different densities), and 20 large instances (10 with $n = 800, l = 40$, and 10 with $n = 1000, l = 50$). Additionally to this testing set, we generate 15 instances, the training set, to perform the calibration of our SO method.

To avoid the overtraining of the SO method, we consider a training set of 5 uniform instances from Stallman's uniform set, and 10 instances from a different source. In particular, we use the `random_bigraph` code of the Standford GraphBase by Knuth (1993) to generate 10 graphs with $n = 100$, and $l = 5$. This set of instances has been employed previously in similar graphs problems (Martí and Laguna, 2003). The entire set of instances (testing and training) is available at http://www.optsicom.es.

The procedures in our method have been implemented in C and the integer linear programming formulation described in Section 2 has been solved using CPLEX 12.6.1. Stallmann's Java source code of his procedures have been downloaded from https://people.engr.ncsu.edu/mfms and have been compiled using Java 8. All the results reported in this section were obtained by running our codes on an Intel Core i7 @ 2.8 GHz and 8GB of RAM computer with the Ubuntu Linux 16.04.LTS – 64bits operating system. We use the following metrics to measure the performance of the methods:

- $Value$: Average objective value of the best solutions obtained by the procedure on the instances considered in the experiment.
- Dev: Average percentage deviation from a reference solution, where the reference solution depends on the testing (i.e., scientific or competitive).
- $Best$: Number of instances in a set for which a procedure is able to match the reference solution, where the reference solution depends on the testing (i.e. scientific or competitive).
- CPU: Average computing time in seconds employed by the algorithm.

5.1 Scientific testing

The goal of this scientific testing is to assess the merit of the elements in our methods as well as to identify the best values for their search parameters. Since each of the following tests isolates these elements, it is not expected that the quality of the solutions obtained by these partial procedures rival those of the best-known (or optimal) solutions that could be found with complete search processes. Therefore, for the purpose of scientific testing, we use as reference solutions in the calculation of Dev the best solutions in each experiment (that the elements being tested are able to produce) in contrast with the best solution known that will be used in the competitive testing reported in the next subsection. This enables the detection of statistical differences between the performance of specific configurations of the elements under study.

In the first experiment, we study the construction methods described in Section 4.1 to calibrate the parameter p, which determines the NCE set. Then in the second experiment, we study the parameter β, which determines the percentage of nodes that is removed in each destruction-construction iteration.

$\boldsymbol{p}$	0.2	0.4	0.6	0.8
Value	76.3	76.2	76.7	78.3
Dev	8.7%	8.7%	9.2%	11.1%
Best	7	8	8	5

Table 2. Calibration of parameter p.

Table 2 shows the *Value, Dev, and Best* statistics corresponding to $p = 0.2, 0.4, 0.6$, and 0.8. Each metric is calculated over the 15 best solutions obtained with the constructive method, one for each instance in the training set. The best solution for a problem instance is selected among 100 solutions generated by each method and parameter value. From Table 2, we can see that the best results of these experiments with respect to *Value*, are obtained with $p = 0.4$. We now search for the most effective value of the parameter β, which controls the number of nodes to be removed after each construction. To that end, we test $\beta = 0.3, 0.5$ and 0.7. Here, the value of p is set to 0.4, according to the previous experiment. Table 3 shows the performance measures associated with each value of β. In this table, we can observe that the procedures with $\beta = 0.5$ and 0.7 obtain the best results in terms of Dev (1.57% and 1.39%, respectively). However, the option of $\beta = 0.5$ clearly obtains the best results in terms of $Best$. So we set β to 0.5.

$\boldsymbol{\beta}$	0.3	0.5	0.7
Value	77.9	76.2	76.4
Dev	3.09%	1.57%	1.39%
Best	4	9	8

Table 3. Calibration of parameter β.

We now hypothesize about the benefits of choosing p randomly, at each iteration of the algorithm, instead of fixing it to a given value for the entire process. This randomization would allow creating NCE sets with different sizes, which could help to diversify the search. In this sense, we have tested three possibilities for this experiment: to keep p fixed to 0.4 (in line with the first experiment), to vary p in the range $[0.4 - 0.1,\ 0.4 + 0.1]$, which allows a controlled randomization, and to leave p freely in $[0, 1]$. Note however that we should be careful with the randomization since very small values of p may produce very large NCE sets that could make the exploration of new positions for its elements a time consuming task.

Table 4 shows the results obtained with these three alternatives where we can see that the worst results in terms of the three metrics are obtained with the second option, $p \in [0.4 - 0.1, 0.4 + 0.1]$, as it obtains higher *Dev* and lower *Best* values than the other options. We can also observe that the random option obtains similar results than the one obtained with the fixed value $p = 0.4$. Since the random option requires a larger computational effort, we set $p = 0.4$ in our Strategic Oscillation method (SO).

$\boldsymbol{p}$	0.4	0.4±0.1	random
Value	76.2	76.7	76.3
Dev	8.7%	8.9%	8.6%
Best	9	6	9

Table 4. Randomization of parameter p.

In the fourth experiment, we generate 1,000 solutions with the SO procedure on a representative instance and compute the two objective functions (max and sum). We undertake to explore if solutions with similar max values ($mc(D)$) have similar sum values ($c(D)$). In the horizontal x-axis of Figure 3, we plotted the values of the max (bottleneck) objective that, in this particular case, range from 175 to 207. In its vertical y-axis, the sum (crossing) values are plotted, ranging from 22,900 to 24,600. Each solution is represented with a point, according to its two objective values. The higher the solution is in a vertical line, the worse is its crossing number, so less efficient they are with respect to the same bottleneck.

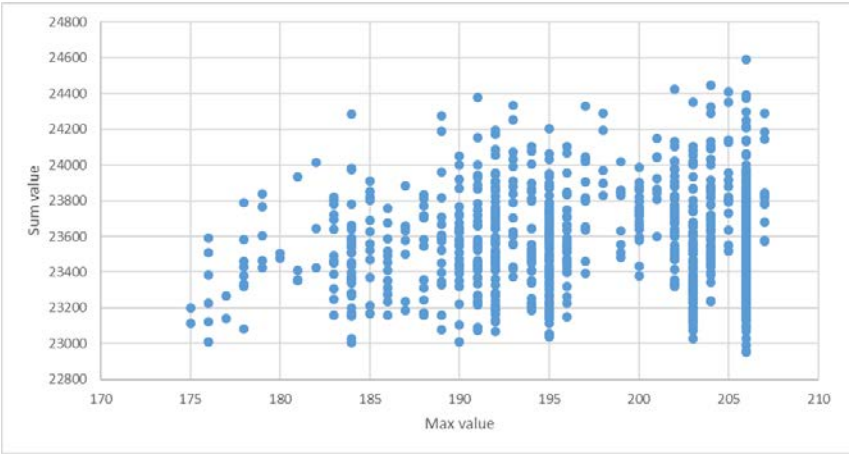

Figure 3. Max and Sum values of 1,000 SO solutions.

Figure 3 shows that for a given value of the max objective function, the method finds solutions with very different values of the sum objective function. This implies that when we solve the min-max problem, if we ignore the min-sum problem, we can obtain solutions with a relatively bad value of this second objective. For example, if we consider in Figure 3 the max value of 195 (depicted in the x-axis), we can see in the y-axis that the sum value of the associated solutions ranges from 23,000 to 24,200, which is relatively wide. This confirms the need of designing a solving method such as the SO that we propose here, that optimize $mc(D)$ as a primary objective, but also optimizes $c(D)$ as a secondary objective. This experiment confirms Stallmann's proposal of optimizing both objectives for a good design of a graph drawing. Otherwise, very inefficient solutions in terms of crossing numbers can be obtained for relatively good values of the bottleneck.

5.2 Competitive testing

As described in Section 1, Martí and Laguna (2003) tested several procedures for the minimization of the sum of crossings $c(D)$. In their study, the authors identified the BC+SW method as a very efficient heuristic to optimize $c(D)$. It is based on constructing solutions using the barycenter and on improving them using a local search procedure with a switching move. This method can be considered a standard in arc crossing minimization, and it has been implemented in many graph drawing systems, starting with the early works by Sugiyama et al. (1981) and Eades and Kelly (1986), to mention a few. We argued that a solution procedure designed for the min-sum is not capable of producing high quality solutions for the min-max. Therefore, a specific algorithm needs to be proposed to solve the latter objective for those applications where the min-max is an important factor. In the following, we provide experimental evidence to support this point.

	MCE			**BC+SW**			**SO**		
Instance	$c(D)$	$mc(D)$	***CPU***	$c(D)$	$mc(D)$	***CPU***	$c(D)$	$mc(D)$	***CPU***
1	280550	267	60	254605	280	1	254075	267	134
2	271078	288	60	246530	279	0	246117	262	61
3	274429	263	60	249272	282	0	249011	270	55
4	283292	269	60	257826	292	0	256924	276	104
5	267934	260	60	242783	272	0	242620	262	65
6	270549	264	60	244775	270	0	244231	259	115
7	276619	287	60	252317	279	0	252114	269	82
8	278208	265	60	252727	287	0	252338	268	74
9	279969	265	60	255047	286	0	254893	269	62
10	277780	265	60	251497	286	0	250908	270	54
Average	276041	269	60	250738	281	0	250323	267	81

Table 5. Results on the two objectives with the three methods

In our first experiment of the competitive testing, we compare our strategic oscillation method, SO, with the BC+SW method implemented by Martí and Laguna (2003), BC+SW, and with the MCE heuristic used by Stallmann (2012). We consider 10 instances from our test set with $n = 300$, density $(m/n) \cong 14$, and 15 layers, all of them generated with Stallmann's generator[2] as

[2] https://people.engr.ncsu.edu/mfms/Software/MinCrossings/MinCrossings.html

described above. We ran the three methods on each instance and evaluated both objectives, min-sum and min-max, on their output solutions. Table 5 shows, for each instance and each method, the $c(D)$ value, $mc(D)$ value and the CPU time in seconds.

Table 5 shows that as expected, BC+SW, designed to optimize $c(D)$, is able to obtain very good results in terms of this objective function, with an average value of 250,738. Since it is a simple heuristic, it requires very short computing times (less than 1 second). However, it is not capable of producing high quality solutions for $mc(D)$ (with an average value of 281). On the other hand, MCE, which focuses on optimizing $mc(D)$, obtains in 60 seconds an average value of this objective of 269 crossings. Despite that this method is supposed to obtain relatively good values for the $c(D)$ objective, it exhibits a poor performance on it, with an average value of 276,041. Our SO method performs remarkably well, with an average $c(D)$ value of 250,323, and an average $mc(D)$ value of 267, achieved on 81 seconds on average.

In the second competitive experiment, we compare MCE and SO with the optimal solution corresponding to minimize $mc(D)$. We apply the mathematical programming formulation described in Section 2, on the well-known IBM CPLEX solver. Table 6 shows the results of this experiment, where each row corresponds to an instance of the problem. Note that due to the enumeration procedure in CPLEX, we can only target small size instances in this experiment (it is indicated in the second column of this table).

In this experiment, we provided the mathematical model to CPLEX using its Callable Library with off-the-shelf configurations, and set a time limit of 3,600 seconds. If the solver is able to provide the optimal value within the time limit, this optimal value and its corresponding CPU time are depicted in columns $mc(D)$ and *CPU*, respectively. Otherwise, we provide the lower and upper bounds that CPLEX reports when the time limit is reached in the form of an interval. Regarding the heuristic methods, Table 6 reports the objective $mc(D)$ value, its deviation with respect to the CPLEX optimal value, the associated computing time, as well as the value of the sum crossings $c(D)$.

		CPLEX		**MCE**				**SO**			
n	m	$mc(D)$	*CPU*	$mc(D)$	*Dev mc*	*CPU*	$c(D)$	$mc(D)$	*Dev mc*	*CPU*	$c(D)$
53	51	2	5	2	0%	60	6	2	0%	1	5
53	50	2	12	3	50%	60	28	2	0%	1	9
59	63	3	49	5	67%	60	46	3	0%	1	11
49	90	[16.7 , 21]	3600	27	-	60	474	25	-	7	384
59	96	12	2551	15	25%	60	267	14	17%	8	227
33	64	12	137	13	8%	60	192	13	8%	2	185
33	69	13	62	15	15%	60	223	15	15%	3	204
55	171	[11.3 , 49]	3600	45	-	60	1846	43	-	28	1682
59	139	[10.49 , 29]	3600	32	-	60	978	31	-	20	879
33	126	32	1476	46	44%	60	1024	37	16%	9	878

Table 6. Comparison with optimal solutions

Table 6 shows that CPLEX is only able to solve seven out of the ten small instances considered. Note that, for the same number of nodes and layers, the mathematical model becomes difficult

to solve when the number of edges increases (i.e., when the graph is denser). See that the interval of the $mc(D)$ values for the three unsolved instances is relatively wide, providing poor information about the optimal values. On the other hand, both heuristics perform considerably well in terms of the $mc(D)$ objective function and CPU time, being SO better than MCE. In particular, MCE exhibits an average percentage deviation of 29.85% on the 7 instances with optimum known, while SO only has an 8%. With respect to the $c(D)$ value, for which we do not know the optimal value, MCE has an average value of 564.22 and SO has an average of 446.40 (note that SO always obtains a better $c(D)$ value than MCE).

We now perform an exhaustive comparison of SO and MCE on different types of instances. Table 7 shows the first results of these final experiments with a comparison on a set of small instances ($n \leq 100$). The first row shows the average results of 10 instances with $n = 60$, number of layers $k = 3$, and density $d \cong 10$. The second group also contains 10 instances, with $n = 100$, $k = 5$, and $d \cong 12$. By columns, for each of the two methods, the table shows the averaged values of both objective functions (columns $c(D)$ and $mc(D)$, respectively) and the CPU time. Note that for each objective the table shows the average value and in brackets the average percentage deviation with respect to the best known value). We calibrate the SO method to run for a similar CPU time than MCE. In particular, we set the maximum number of iterations to 5000, and an additional criterion that if in 1000 consecutive iterations the best solution does not improve, the method stops.

			MCE			**SO**		
$\boldsymbol{n}$	$\boldsymbol{k}$	$\boldsymbol{d}$	$\boldsymbol{c(D)}$	$\boldsymbol{mc(D)}$	***CPU***	$\boldsymbol{c(D)}$	$\boldsymbol{mc(D)}$	***CPU***
60	3	10	37765 (9%)	247 (3%)	60	34821 (0%)	242 (1%)	32
100	5	12	76644 (9%)	254 (0%)	60	70594 (0%)	255 (1%)	69
Average			57204 (9%)	250 (2%)	60	52708 (0%)	248 (1%)	50

Table 7. Comparison of the two methods on 20 small size instances.

Table 7 shows that our SO method performs very well on the small instances considered, obtaining results that exhibit an average $mc(D)$ value slightly better than MCE (with the exception of some instances where MCE obtains the best values). Regarding the sum of crossings, $c(D)$, Stallmann's MCE algorithm performs poorer, obtaining an average deviation of 9%, while SO exhibits a remarkable 0%. We perform a Wilcoxon non-parametric test for paired samples to compare the results of both methods. Considering the $mc(D)$ values, we obtain a p-value of 0.2 which indicates that, in general, there are no significant differences between the methods with respect to this objective. On the contrary, when we consider the $c(D)$ values, the p-value of this test is 0.000089, which confirms the superiority of the SO method in terms of the min-sum objective.

In the experiment shown in Table 7 we considered instances with a density value close to 12. In the next experiment we undertake to compare both methods for different values of the instances density. In particular, we solve 30 mid-sized instances ($n = 300$) divided into three groups, each one with 10 instances and a density value equal to 9.4, 14.1 and 16.8 on average respectively. Table 8 summarizes the results of both heuristics, where each row shows average results over the 10 instances in each group.

	MCE			SO		
Density (d)	$c(D)$	$mc(D)$	**CPU**	$c(D)$	$mc(D)$	**CPU**
9.4	116369 (18.6%)	168 (0.0%)	60	98696 (0.0%)	173 (2.9%)	70
14.1	276041 (10.3%)	269 (1.9%)	60	250323 (0.0%)	267 (1.1%)	81
16.8	402368 (5.9%)	328 (0.5%)	60	380061 (0.0%)	331 (1.6%)	250

Table 8. Comparison of the two methods on medium size instances

Table 8 shows that MCE is able to obtain the best results for the min-max objective, $mc(D)$, since it presents lower average deviations than SO. This has an exception in two instances with density 14.1 in which SO obtains better values, which causes the average deviation of MCE to increase to 1.9%. In general terms we can say that both methods present on average similar results, being MCE slightly better. We cannot identify a different performance depending on the density of the instances. Regarding the min-sum objective, $c(D)$, it is clear that SO performs much better than MCE, since the former always obtains the best known solution. The Wilcoxon test on the results of this table exhibits similar results than those obtained for the previous table: a p-value > 0.05 for the $mc(D)$ objective, and a p-value < 0.05 for the $c(D)$ objective.

In our last competitive experiment, we test the two methods and compare their results on 30 large instances, ($500 \leq n \leq 1000$, and up to 50 layers). We ran the methods for longer (and similar) running times for these large instances. Table 9 shows the results of this experiment.

			MCE			SO		
n	k	d	$c(D)$	$mc(D)$	**CPU**	$c(D)$	$mc(D)$	**CPU**
500	25	14.5	475970 (11%)	270 (1%)	60	429656 (0%)	273 (2%)	146
800	40	9.8	334285 (20%)	175 (0%)	300	278458 (0%)	181 (3%)	313
1000	50	9.8	421128 (20%)	176 (0%)	300	349641 (0%)	183 (4%)	476
Average			410461 (17%)	207 (0%)	220	352585 (0%)	212 (3%)	312

Table 9. Comparison of the two methods on large size instances

Each row in Table 9 corresponds to a set of 10 instances with characteristics described in the first three columns. We can see that Stallmann's method, MCE, is able to achieve slightly better results than our proposed method SO in terms of maximum crossings on large size instances. In particular MCE presents a remarkable 0% while SO has a 3% of average deviation with respect to the best known $mc(D)$ values. However, with respect to the secondary objective $c(D)$, SO clearly outperforms MCE since it always matches the best known solution, and MCE presents an average percent deviation of 17%. To sum it up, SO is able to compete with MCE in terms of minimizing the maximum number of crossings and improves it when minimizing the total (sum) number of crossings, thus it constitutes a very good method to generate readable drawings.

5.3 Graph Drawing Expert Systems

In this subsection we complement the experimentation above by drawing the output of our method and comparing it with those obtained with well-known graph drawing systems. In particular, we apply the SO method to the example depicted in Figure 2 in an arbitrary node ordering. We can see that In Figure 2 $mc(D) = 73$ and $c(D) = 2647$, which makes the drawing very difficult to analyze. Figure 4 shows the output of our method on the same example. It is a much more readable drawing, where $mc(D) = 36$ and $c(D) = 1675$. It is still a complex drawing since the graph has 255 arcs; but there is no doubt that the solution of the min-max problem is simply better to capture the graph information.

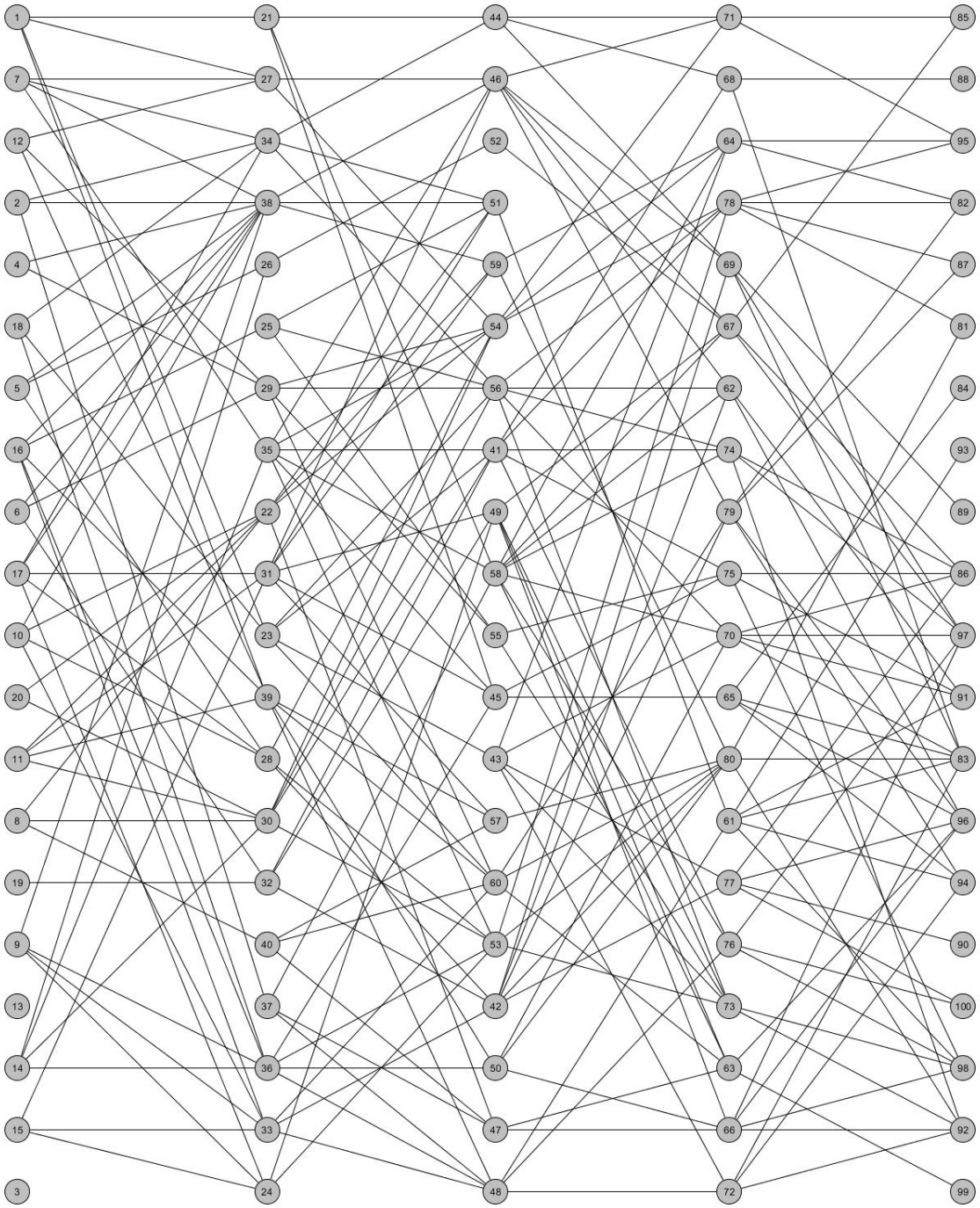

Figure 4. Optimized drawing with our method of the example in Figure 2.

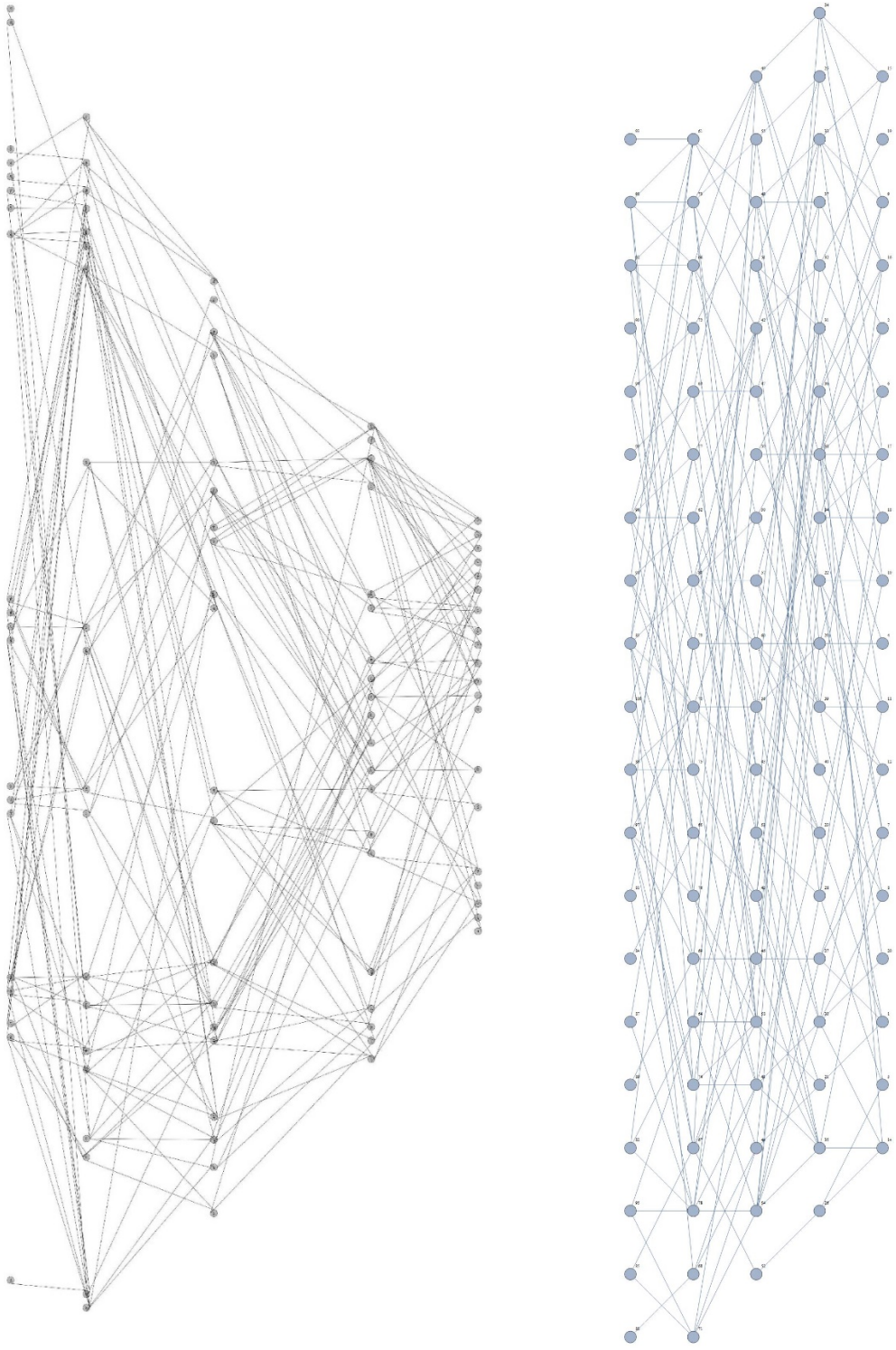

Figure 5. Optimized drawing with *yEd* (left) and *Mathematica* (right) of the example in Figure 2.

We now consider two graph drawing expert systems among those listed in Table 1. Note that all of them implement simple rules for crossing reduction (based on the barycenter or median heuristics). Specifically, we apply *yEd* and *Mathematica* to the same graph represented in Figures 2 and 4, and analyze the automatic drawings obtained with these systems in terms of arc crossings. Although *yEd* does not provide a direct evaluation of the number of crossings, we have identified in its output represented in Figure 5 (left) that arc (22 , 47) has more than 60

crossings. Mathematica provides a tool for arc crossing computation, which returns for this drawing (Figure 5 right) the maximum value $mc(D) = 49$, and the sum value $c(D) = 1822$. Therefore, regarding the maximum number of crossings among all its arcs, we can conclude that these drawings are worse than our proposal shown in Figure 4, which has a maximum crossing value of 36.

We also address an interesting point in the context of graph drawing systems: the influence of the graphic standard on the number of crossings. Although in this paper we restrict our attention to hierarchical (or layered) drawings, it is indeed interesting to compare this drawing convention with another representation on the same graph. To this end, we run *yEd* to obtain a circular layout of the graph depicted in the previous figures. The result, shown in Figure 6, confirms that in terms of arc crossing the hierarchical layout is a good choice, since the circular layout contains, in general, many more crossings. For example, arc (22, 59) in Figure 6 has close to 100 crossings, indicating that its maximum number of crossings, $mc(D)$, is larger than 100, and therefore this can be considered much worse than our layered drawing shown in Figure 4 with $mc(D) = 36$.

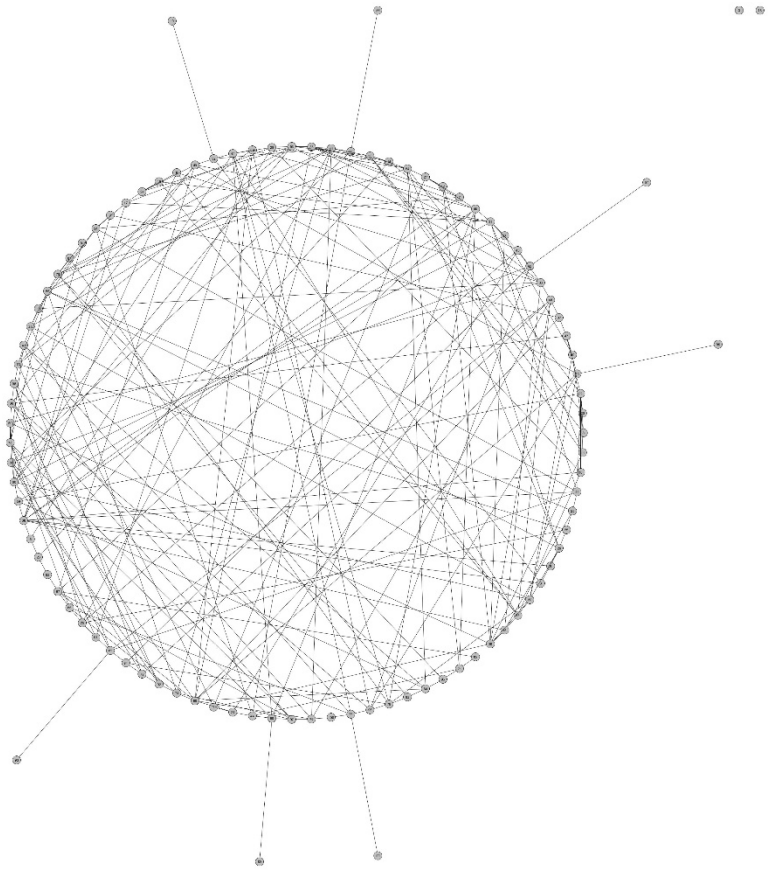

Figure 6. Circular drawing obtained with *yEd* Expert System of the example in previous figures.

6. Conclusions

In this paper, we investigate the adaptation of the Greedy Randomized Adaptive Search Procedure (GRASP) and Strategic Oscillation (Iterated Greedy) methodologies to the Graph Drawing Problem. In particular, we focus on the effect of the balance between randomization

and greediness on the performance of these multi-start heuristic search methods when solving this NP-hard problem.

We target here a very interesting variant in arc crossing minimization recently proposed. We consider two objective functions, the min-max, as a primary objective, and the min-sum, as a secondary objective. Our proposal is able to compete with the previous method in the primary objective and outperforms it in the secondary one. In practical terms our achievements are very appealing since they show that our method is able to provide very good drawings.

After exhaustive experimentation we can conclude that our SO method obtains solutions of better quality (lower number of arc crossings) than those obtained with existing methods. This comparison includes the recent MCE method for crossing reduction and two expert systems for graph drawing (yEd and Mathematica). Note however, that when solving large instances, our method requires slightly longer computational times than MCE. This was expected since advanced search mechanisms require greater computational effort than simple ordering rules in which most of the previous methods rely. A direct application of our findings is to implement the SO method in commercial graph drawing expert systems to improve their functionality in terms of crossing reduction. Nowadays computers permit to execute complex metaheuristics in reasonable CPU time, thus expert systems can replace its simple heuristic rules, such as the barycenter or the median, with efficient metaheuristics such as the SO proposed here.

A natural extension of our work is to apply metaheuristic methodologies to solve other optimization problems in the context of graph drawing. In particular, within the layered standard that we consider in this paper, the first step of the so-called Sugiyama's method is to assign vertices to layers. Graph drawing expert systems referenced here perform this task with a well-known method, the Coffman-Graham layering, obtaining relatively good solutions in short computational time. However, the layering problem is indeed NP-hard, and we believe that better solutions can be obtained with modern metaheuristics. This also holds for other optimization problems in the drawing process, such as cycle removal, dummy vertices minimization, or coordinate assignment.

Acknowledgements

This work was supported by the Spanish Ministerio de Economía y Competividad and Fondo Europeo de Desarrollo Regional (MINECO/FEDER) (TIN-2015-65460-C02-01, project MTM-2015-68097, and PhD. grant BES-2013-064245). Authors wish to thank Prof. Stallmann for providing us with the MCE code and instances to perform the experiments reported in this paper. Authors also want to thank the anonymous referees to help them to connect their findings with graph drawing expert systems.

References

Bachmaier, C., F. J. Brandenburg, W. Brunner, and F. Hübner (2010). "A global k-level crossing reduction algorithm." In "Proceedings of the 4th International Workshop on Algorithms and Computation (WALCOM'10)". *Lecture Notes in Computer Science* 5942, 70-81.

Bhatt, S. N. and F. T. Leighton (1984). "A Framework for Solving VLSI Graph Layout Problems". *Journal of Computer and System Sciences* 28, 300-343.

Carpano, M. J. (1980). "Automatic Display of Hierarchized Graphs for Computer Aided Decision Analysis", *IEEE Transactions on Systems, Man, and Cybernetics* 10 (11), 705-715.

Corberán, A., J. Peiró, F. Glover, V. Campos, and R. Martí (2016). "Strategic Oscillation for the capacitated hub location problem with modular links". *Journal of Heuristics* 22 (2), 221-244.

Di Battista, G. D., P. Eades, R. Tamassia, and I. G. Tollis (1999). "Graph Drawing. Algorithms for the Visualization of Graphs". Prentice Hall, New Jersey.

Eades, P. and D. Kelly (1986) "Heuristics for Drawing 2-Layered Networks". *Ars Combinatoria* 21, 89-98.

Festa, P. and M. G. C. Resende (2011). "GRASP: basic components and enhancements". *Telecommunication Systems* 46 (3), 253–271.

Garey, M.R. and D.S. Johnson (1983) "Crossing number is NP-complete". *SIAM Journal on Algebraic Discrete Methods* 4, 312-316.

Glover, F. and M. Laguna (1997). "Tabu search". Kluwer, Norwell, MA.

Himsolt, M. (1996). "GraphEd: an interactive graph editor" , Proc. STACS 89 (349) of Lecture Notes in Computer Science , Berlín, 532-533.

Jünger, M. and P. Mutzel (1997). "2-Layer Straightline Crossing Minimization: Performance of Exact and Heuristic Algorithms". *Journal of Graph Algorithms and Applications* 1(1), 1-25.

Jünger, M. and P. Mutzel (2004). "Graph drawing software", Springer, Berlín.

Knuth, D. E. (1993). "The Stanford GraphBase: A platform for combinatorial computing". Addison Wesley, New York.

Laguna M. and R. Martí (1999). "GRASP and Path Relinking for 2-Layer Straight Line Crossing Minimization". *INFORMS Journal on Computing* 11 (1), 44-52.

Laguna M., R. Martí and V. Valls (1997) "Arc Crossing Minimization in Hierarchical Digraphs with Tabu Search", *Computers and Operations Research* 24 (12), 1175-1186.

Martí, R. and M. Laguna (2003). "Heuristics and Meta-Heuristics for two layer straight line crossing minimization". *Discrete Applied Mathematics* 127 (3), 665-678.

Martí, R., M. Resende, and C. Ribeiro (2013). "Multi-Start Methods for Combinatorial Optimization". *European Journal of Operational Research* 226, 1-8.

Matuszewski, C., R. Schönfeld, and P. Molitor (1999). "Using sifting for k-layer straightline crossing minimization". In "Proceedings of the 8th International Symposium on Graph Drawing". *Lecture Notes in Computer Science* 1731, 217-224.

Ruiz, R., and T. Stützle (2008). "An iterated greedy heuristic for the sequence dependent setup times flowshop problem with makespan and weighted tardiness objectives". *European Journal of Operational Research* 187 (3), 1143-1159.

Rowe, L.A., M. Davis, E. Messinger, C. Meyer, C. Spirakis, and A. Tuan (1987) A browser for directed graphs", Software, Practice and Experience 17(1), 61-76.

Stallmann, M.F. (2012) "A Heuristic for Bottleneck Crossing Minimization and Its Performance on General Crossing Minimization: Hypothesis and Experimental Study". *ACM Journal of Experimental Algorithms* 17(1), 1 – 30.

Sugiyama, K., S. Tagawa, and M. Toda (1981) "Methods for Visual Understanding of Hierarchical System Structures". *IEEE Transactions on Systems, Man, and Cybernetics* SMC-11 (2), 109-125.