# Heuristics for the Constrained Incremental Graph Drawing Problem

Antonio Napoletano[1], Anna Martínez-Gavara[2], Paola Festa[1], Tommaso Pastore[1], and Rafael Martí[2]

[1]Department of Mathematics and Applications, University of Napoli Federico II, Compl. MSA, Via Cintia, 80126, Napoli, Italy.
{antonio.napoletano2, paola.festa}@unina.it
[2]Department of Statistics and Operations Research, University of Valencia, Dr. Moliner 50, 46100 Burjassot, Valencia, Spain
{gavara, rafael.marti}@uv.es

October 11, 2018

**Abstract**

Visualization of information is a relevant topic in Computer Science, where graphs have become a standard representation model, and graph drawing is now a well-established area. Within this context, edge crossing minimization is a widely studied problem given its importance in obtaining readable representations of graphs. In this paper, we focus on the so-called incremental graph drawing problem, in which we try to preserve the user's mental map when obtaining successive drawings of the same graph. In particular, we minimize the number of edge crossings while satisfying some constraints required to preserve the position of vertices with respect to previous drawings. We propose heuristic methods to obtain high-quality solutions to this optimization problem in the short computational times required for graph drawing applications. We also propose a mathematical programming formulation and obtain the optimal solution for small and medium instances. Our extensive experimentation shows the merit of our proposal with respect to both optimal solutions obtained with `CPLEX` and heuristic solutions obtained with `LocalSolver`, a well-known black-box solver in combinatorial optimization.

**Keywords:** heuristics, metaheuristics, combinatorial optimization, graph drawing.

## 1 Introduction

Graph drawings are difficult to analyze when their elements (vertices and edges) are placed in an unorganized way, ignoring basic aesthetic criteria. The graph drawing problem consists in creating a representation of a given graph in such an organized way that it is easily readable. The range of topics in the graph drawing field goes from visual perception to algorithms or models and it currently constitutes a well-established area in Computer Science (see for example Kaufmann and Wagner [24]). The conventional wisdom that a picture is worth a thousand words can be adapted here by adding that it is even better if the picture is generated automatically rather than manually.

The graph drawing literature (see [3]) reports several standards to represent the vertices, such as dots, circles, or boxes, and more importantly, to represent the edges: straight lines, polygonal and/or orthogonal paths, or arbitrary curves. Graph drawing resources (papers, software, conference proceedings) are usually classified according to these different types of representations, also called drawing conventions. In this paper, we focus on hierarchical representations, which are also known as layered graphs, where directed acyclic graphs are represented by arranging the vertices on a series of equidistant vertical lines, called layers, in such a way that all edges, drawn as straight lines, point in the same direction. It should be noted that working with hierarchical representations does not limit the scope of our work since there are several methods to transform a directed acyclic graph into a hierarchical graph, the most well-known being the procedure in Sugiyama et al. [40]. Table 1 shows some applications of hierarchical graphs.

| **Context** | **References** | **Data represented** |
|---|---|---|
| Workflow visualization | [41] | Work to be executed by the project team. |
| Software engineering | [10, 8] | Calling relationships between subroutines in a computer program. |
| Database modeling | [21] | Data connection, processing and storing inside the system. |
| Bioinformatics | [28] | Structured molecules with multiple functional components. |
| Process modeling | [20, 14] | Analytical representation or illustration of an organization's business processes. |
| Network management | [35, 26] | Set of productive actions. |
| VLSI circuit design | [4] | Design of integrated circuits (ICs) of new semiconductor chips. |
| Decision diagrams | [37, 33] | Logic synthesis and circuits. |

Table 1: Some hierarchical graph applications.

Although generally speaking the perception of the quality of a graph representation is fairly subjective, edge crossing minimization is a well-recognized method to obtain a readable drawing. Other aesthetic criteria to capture the notion of readability include: bend, length and area minimization, angle maximization, and symmetries and clustering (see [24] for a detailed description). In this paper, we target edge crossing minimization because of its importance as an aesthetic criterion and its difficulty in terms of heuristic optimization (it is an NP-Complete problem, even when the graph only has two layers, as shown in [18]). The problem of minimizing the edge crossings in a Hierarchical Directed Acyclic Graph (HDAG) has been typically addressed as the problem of finding the optimal ordering of vertices in each layer ([29]).

Graphs have become a fundamental modeling tool in several fields, such as project management, production planning or CAD software, where changes in project structure result in successive drawings of similar graphs. The so-called mental map of a drawing reflects the user's ability to create a mental structure with the elements in the graph. When elements are added to or deleted from a graph, the user has to adjust his/her mental map to become familiar with the new graph. The dynamic graph drawing area is devoted to minimizing this

effort. As described in [6], considering that a graph has been slightly modified, applying a graph drawing method from scratch would be inefficient and could provide a completely different drawing, thus resulting in a significant effort for the user to re-familiarize him/herself with the new map. Therefore, models to work with dynamic or incremental graphs have to be used in this context.

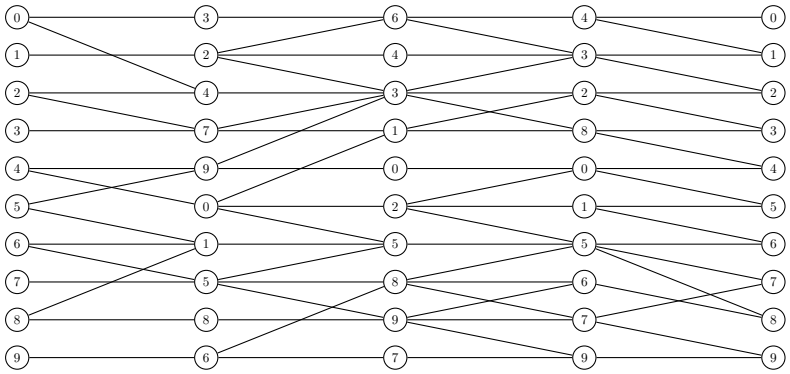

Figure 1: Optimal drawing for crossing minimization of a given graph.

To illustrate the incremental problem, we consider the hierarchical drawing in Figure 1, which shows the optimal solution of the edge crossing minimization problem of a graph with 50 vertices and 5 layers. It has been obtained with CPLEX by solving the classical mathematical formulation of the problem shown in Section 3. We increment this graph now by adding 20 vertices (4 in each layer), and their incident edges. Figure 2 shows the optimal solution of the edge crossing minimization problem of the new graph, where the new vertices and edges are represented with dotted lines.

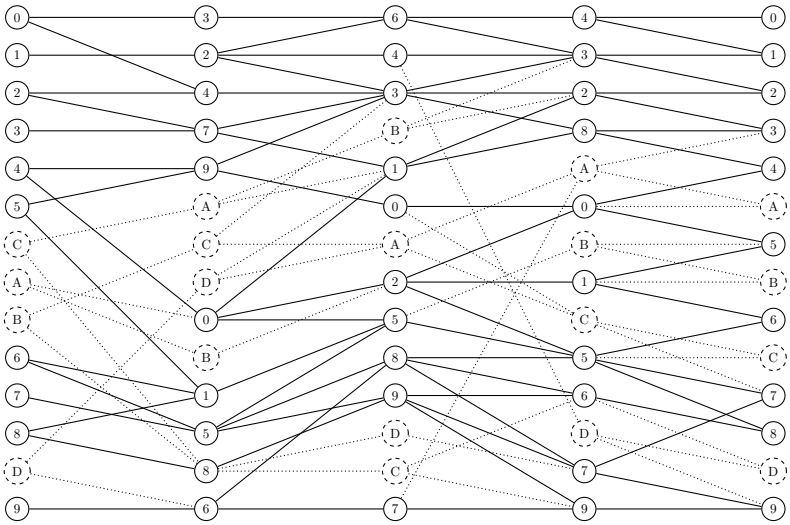

Figure 2: Optimal drawing for crossing minimization of the incremented graph.

Although the number of crossings in Figure 2 is minimum, 99, this new drawing was created from scratch, and it ignores the position of the vertices in the original drawing of Figure 1. For example, vertex 6 in the first layer is in position 7 in Figure 1, but in position

10 in Figure 2. We can say that Figure 2 does not keep the mental map of the user familiarized with Figure 1. Therefore, in line with the dynamic drawing conventions [6], we propose to reduce the number of crossings of the new graph while keeping the original vertices close to their positions in Figure 1.

The literature on incremental graph drawing is scarce. We report in the next section the main contributions of its few papers. In particular, regarding hierarchical graphs, previous efforts only preserve the relative positions of the original vertices [30]. As it will be shown, this may result in poor incremental drawings (i.e., the mental map of the drawing is not properly kept). In this paper, we consider a robust model with constraints on both the relative and the absolute position of the original vertices when minimizing the number of edge crossings in a sequence of drawings. In this way, we help the user to keep his or her mental map when working with a drawing where successive changes occur. In particular, our model restricts the relative position of the original vertices with respect to their position in the initial drawing (as in [30]), and also restricts their absolute position within a short distance of their initial position (as in [3] in the context of orthogonal graphs).

As far as we know, this is the first time that both constraints are included in incremental drawings in hierarchical graphs. In this paper, we propose a mathematical programming formulation to obtain the optimal solution for medium-sized instances. We also propose heuristic methods, based on GRASP and Tabu Search, to obtain high-quality solutions for large-size instances. Additionally, we adapt the well-known LocalSolver black-box optimizer to solve this problem. We perform an empirical comparison with these four methods.

The study reported herein goes beyond solving a particular optimization problem. Tabu Search and GRASP can be considered as representative or even flagship procedures of two important classes of metaheuristic methodologies: memory-based and memory-less methods. The explicit use of memory structures constitutes the core of a large number of intelligent solvers, including Tabu Search ([19]) or Path-Relinking ([38]). On the other hand, we can also find successful metaheuristics, such as Simulated Annealing ([25]) or GRASP ([15]), with no memory structure in their original designs. Following previous papers on heuristic optimization ([31]), here we try to answer an open question in the field: is the use of memory a good idea or is it simply better to resort to semirandom designs? In this paper, we compare both designs, memory-based and semirandom, when solving the incremental graph drawing problem.

Considering that we are introducing a new problem, we motivate it and revise similar problems in Section 2. Then, we propose a mathematical programming formulation in Section 3. The main contributions of our paper are described in Section 4, where we apply metaheuristic methodologies to obtain efficient solutions for this problem. The paper finishes with an extensive computational experimentation to assess the merit of our methods, and with the associated conclusions in Section 7.

## 2 Motivation and Problem Background

It is well documented that incremental drawing is a very important area in graph representations. We can find many references highlighting this problem, or more precisely, this family of problems. We refer the reader to [34], [11], or [36], to mention a few. The graph drawing textbook [3] is a reference in the field, and it devotes an entire chapter to incremental constructions. The range of applications of incremental techniques is also vast, from on-line

problems, such as affiliation networks or on-line advertisement, to the well-known project management diagrams in business administration. However, in spite of its importance and practical significance, there are just a few incremental graph drawing models, and in our opinion, they are not entirely satisfactory. In this section, we review the incremental graph drawing background, including previous efforts in both software and academic context. We discuss the limitations of existing models for hierarchical drawings and how our proposal overcomes them.

In affiliation networks, individuals and groups are depicted with vertices, and edges represent the membership of individuals to those groups. These networks usually change in time, since new groups and members are systematically added. When these additions occur, it is desirable that the new layout is both aesthetically pleasing and preserves dynamic stability (i.e., it stands well into the sequence of drawings). A similar situation can be found in queries on on-line advertisement [2], which have to be represented as a sequence of graphs for their analysis. A link between a query and an advertisement (ad) indicates that the query has been used to reach that specific ad. Graphs of this kind are naturally dynamic, since users are continuously submitting queries, and new ads are also included by the companies.

Currently, there exists a wide variety of software devised for graph representation. For example, Graphviz [17] is a free, flexible software, accessible with an easy-to-use web version. As most of its competitors, it incorporates optimization criteria in order to obtain aesthetically pleasing drawings. We illustrate this point in Figure 3, which shows how Graphviz is able to obtain a clear and aesthetically pleasing layout for a simple hierarchical graph.

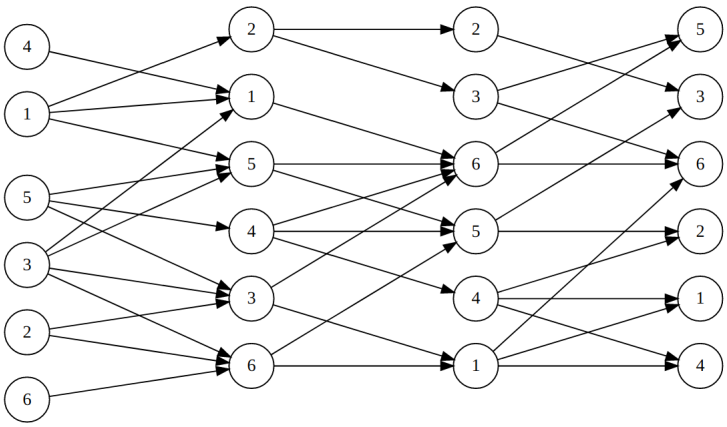


Figure 3: Graphviz drawing of a simple hierarchical graph.

However, this software is not able to properly represent objects and relations characterized by dynamic nature. Indeed, whenever new vertices and edges are added to the network, the software does not support the identification of incremental elements, and it draws the graph from scratch. For example, in Figure 4, also obtained with Graphviz, we can observe how the addition of a new set of vertices and connections in the graph of Figure 3 results in a completely different representation, thus "destroying" the mental map of the reader.

Recently, in the scientific literature, some efforts were carried out in order to solve incremental graph drawing problems. The heuristic algorithms proposed are able to solve large instances, but the resulting drawings present shortcomings in terms of mental maps, as it can be observed in the following example arising from *project management*. In these projects,

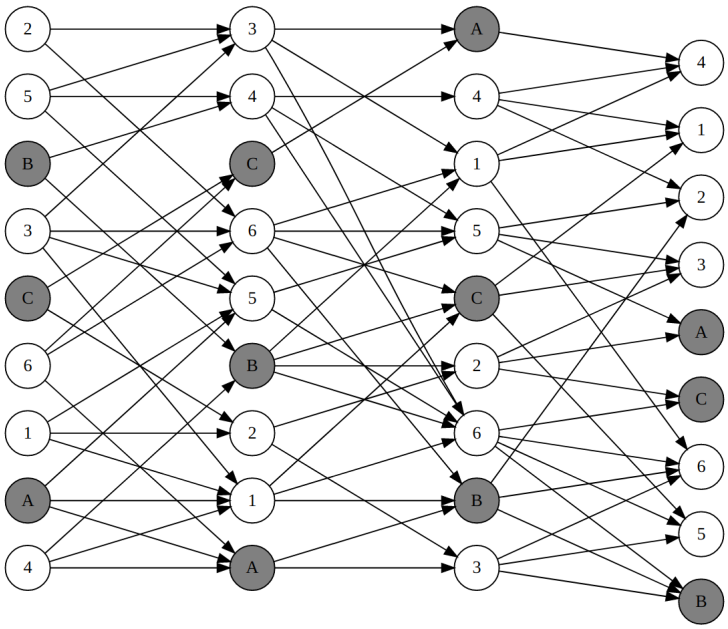


Figure 4: Graphviz drawing of an incremental hierarchical graph.

tasks are represented with vertices and edges model their precedence relationships. Many changes occur during the development of a large project and they have to be reflected in the associated graph or chart. Dynamic graph drawing is a demand of project managers, who need a stable sequence of drawings as the project evolves. The project is usually represented as a hierarchical or layered graph, and it constitutes a good example of the applicability of our incremental graph drawing model. Figure 5 shows a representation of such a graph on a medium size project. Since it is a large graph with 6 layers, we made some simplifications to draw it. In particular, we do not include the first vertex, which represents the beginning of the project. It would be allocated in the left part (say in layer 0) and connected to all the vertices in the first layer. Similarly, we do not represent the final project's vertex, which would be allocated in layer 7, and connected with an edge to each vertex in layer 6. To reduce the size of the vertices in the drawing, we renumbered them, starting from 1 in each layer. We color in light gray the original vertices and edges, which were in the initial design of the project, and with black the new vertices and edges that have been added in a later stage.

In Figure 5, the new vertices are all placed in the bottom part of the diagram, leaving in this way the original vertices in their initial position. This is good in terms of the stability of the drawing. In other words, it preserves the reader's mental map of the project. However, it contains a large number of crossings, since no optimization has been performed after the addition of the new vertices. This drawing has 6963 edge crossings. The challenge is therefore to reduce the number of crossings while trying to keep the placement of the original vertices as much as possible. That is essentially the objective of the incremental graph drawing problem.

We have identified two types of approaches in dynamic graph drawing algorithms with the objective of creating a sequence of stable representations, as introduced by Böhringer and Paulisch in [5]. The first type consists of multi-objective methods, which optimize both the aesthetic criteria and a stability distance function. In this category, we can find the early work by North in [34], who proposed a graph drawing system; Branke in [6], who adapted

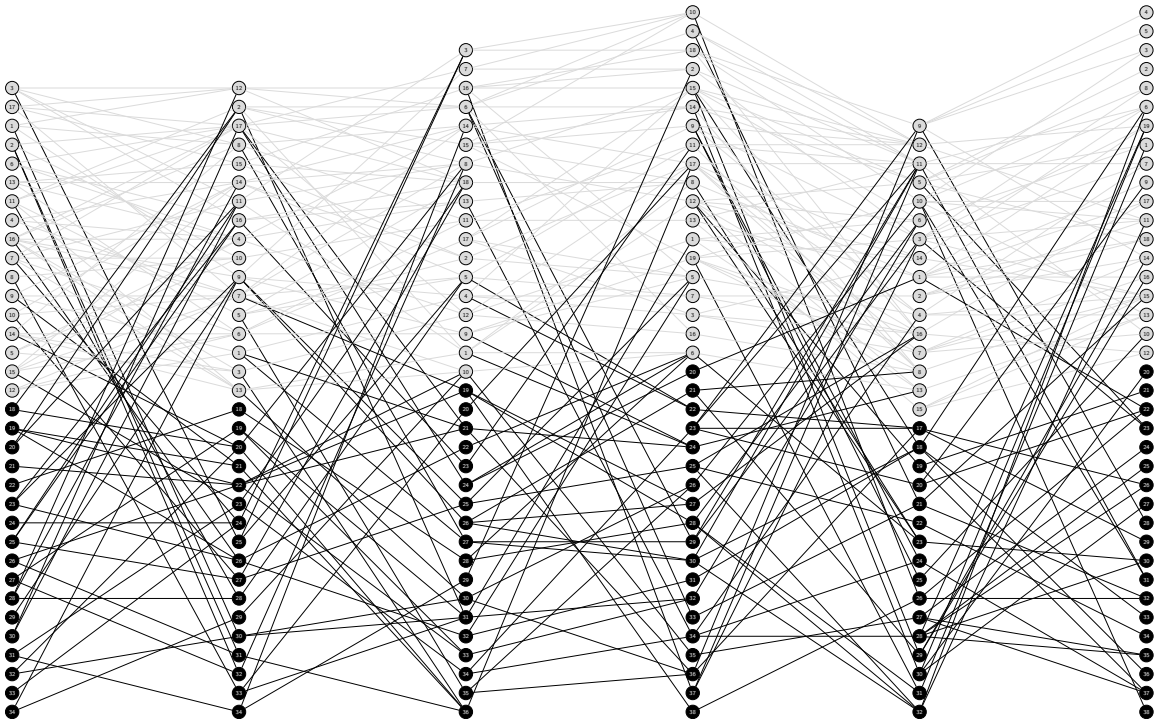

Figure 5: Project management example.

Sugiyama's heuristic to include the stability conditions in the drawing method; and Pinaud et al. in [36], who based their stability measure in terms of the number of pairs of vertices that are inverted in the new drawing with respect to their relative position in the previous drawing.

In the second type of approaches, called incremental graph drawing algorithms, we can find the models based on the inclusion of additional constraints in the standard crossing reduction problem. In these approaches, hard constraints restricting the position of the vertices in the new graph are established beforehand, as opposed to the guidelines (soft constraints or objective functions) considered in the first type described in the previous paragraph. In particular, Martí and Estruch considered in [29] the relative position between vertices in the original drawing as a constraint to create a new incremental drawing when new vertices and edges are added to the bipartite graph (their GRASP approach was limited to optimize 2-layered graphs). Recently, Martí et al. proposed in [30] a Tabu Search for the same problem obtaining better solutions than the previous GRASP. Then, in [39], these authors extended this approach to the general case of multi-layer graphs and proposed a new heuristic based on the Scatter Search methodology, which outperforms the previous GRASP. We apply this method to the example in Figure 5, obtaining the drawing shown in Figure 6 with 4647 edge crossings.

The methods described above, and applied in Figure 6, solve the incremental or dynamic problem based on the relative position of the original vertices. These vertices, depicted in gray color, keep the same relative ordering than in the original drawing shown in Figure 5. However, we believe that this example also illustrates a serious drawback of this model based on the relative ordering: the location of the original vertices changes significantly, thus

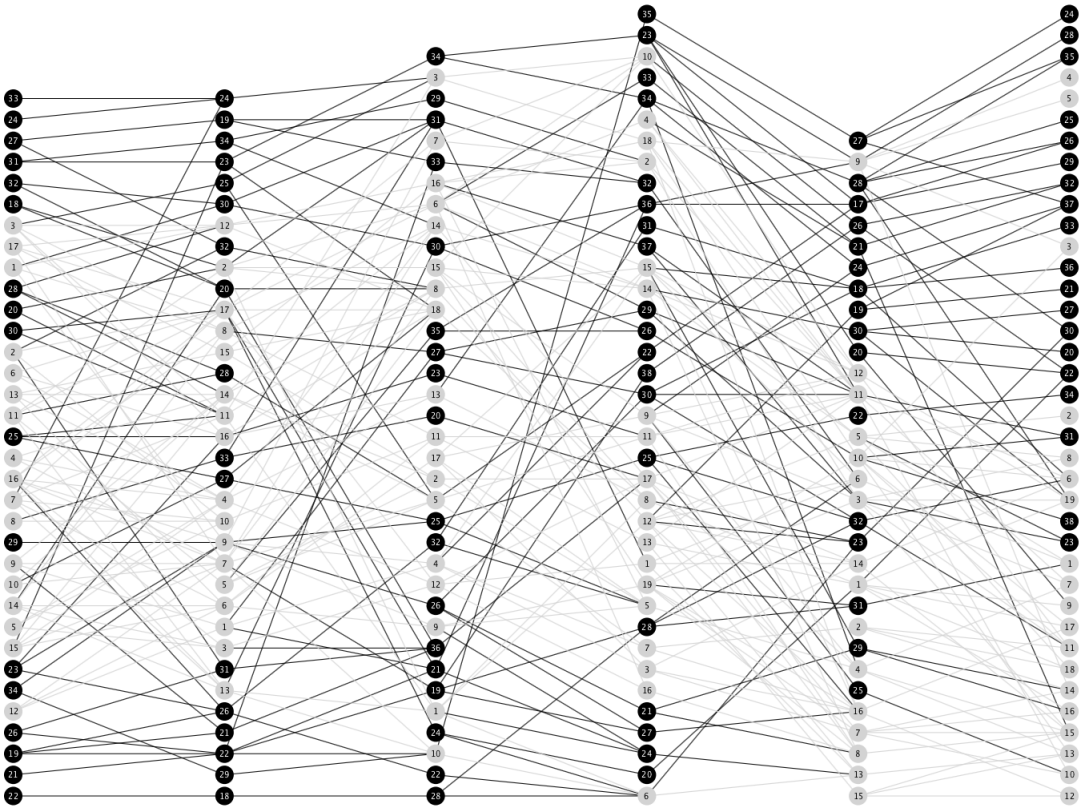

Figure 6: Example optimized with a previous method.

endangering the user's mental map of the original drawing. For example, in the drawing of Figure 5, vertex 3 of the first layer is in the first position, while in the drawing of Figure 6, it is in the seventh position. This is due to six new vertices (33, 24, 27, 31, 32, and 18) inserted in previous positions to reduce the number of crossings. Similar and even worse situations can be easily identified in many original vertices that are now (in Figure 6) in positions far from their original placement (shown in Figure 5). If for example we consider the last original vertex in layer 1, number 12, which occupied position 17th in Figure 5, we can see that it is now (in Figure 6) in position 30. We believe that this difference of 13 positions alters the layout too much and forces the user to make important adjustments in his or her mental map of the graph, thus making the reading of the graph a time consuming task. In Section 6, to complement the experimentation, we will show the solution of our method on the example depicted in Figure 5.

An in-depth study of the relations of the mental map and several formal measures can be found in [7], in the case of orthogonal graphs. The authors define mathematical criteria to reflect the idea of stability across drawings based on the comparison of different layouts. In particular, they define measures and conduct a student survey to evaluate them. The authors conclude this study with the analysis of the correspondence between theoretical results and the survey, thus leading to a ranking of the measures defined, where among the most important are the relative distance, the nearest neighbor-within position, and the average and maximum distances.

Considering that the method based on the relative position [30] is somehow limited, we

approach the stability of hierarchical graphs taking into account both relative and absolute distances of the vertices position by including additional constraints. Our proposal is in line with previous studies. In the case of orthogonal graphs ([7]), the authors consider different measures based on the distances to achieve stability. Similarly, Di Battista et al. [3] proposed to approach the incremental problem with the *coordinates scenario*, in which the coordinates of some vertices and edges may change by a small constant, because of the insertion of a new vertex and its incident edges. Branke [6] pointed out that it is not possible to state that relative position is better than absolute position in terms of preserving the user's mental map, and both can be of interest.

To sum up, based on the previous studies and the examination of practical examples, we propose a new model to preserve the mental map of incremental hierarchical drawings, while minimizing the number of crossings. In particular, we fix the relative position among the original vertices, and keep their absolute position within a short bounded distance $K$ of their original positions.

## 3 Mathematical Programming Models

Crossing minimization is a classic problem in graph drawing. Given a graph $G = (V, E)$, the goal is to find an ordering of their vertices with the minimum number of edge crossings. We focus on multi-layered hierarchical graphs $H = (V, E, p, L)$, where $V$ and $E$ represent the sets of vertices and edges, respectively, $p$ is the number of layers, and $L : V \to \{1, 2, \ldots, p\}$ is a function that indicates the layer where a vertex $v \in V$ resides. Let $V^t$ be the set of vertices in layer $t$, i.e., $V^t = \{v \in V : L(v) = t\}$, and let $E^t$ be the set of edges from $V^t$ to $V^{t+1}$. Given a vertex $v$ in layer $t$, let $\Lambda_{t-1}(v) = \{u \in V^{t-1} : (u, v) \in E\}$ be the set of vertices in layer $t-1$, adjacent to $v$. Symmetrically, let $\Lambda_{t+1}(v) = \{u \in V^{t+1} : (v, u) \in E\}$ be the set of vertices in layer $t+1$, adjacent to $v$, and then $\Lambda(v) = \Lambda_{t-1}(v) \cup \Lambda_{t+1}(v)$. We define a drawing $D$ as a pair $(H, \Pi)$, where $H$ is a hierarchical graph and $\Pi$ is the set $\{\pi^1, \ldots, \pi^p\}$, with $\pi^t$ establishing the ordering of the vertices in the layer $t$. Let $C(D)$ be the number of edge crossings in drawing $D$.

Jünger and Mutzel proposed in [22] a linear integer formulation for the Bipartite Drawing Problem (BDP). A generalization of this approach for the Multi-Layer Graph Drawing Problem is proposed by Jünger et al. in [23]. In the multi-layer case, we can denote the set of vertices and edges as $V = V^1 \cup \ldots \cup V^p$ and $E = E^1 \cup \ldots \cup E^{p-1}$, in such a way that edges are only allowed between successive layers, with $n_t = |V^t|$. To formulate the integer linear problem, the authors define in [23] the binary variables $x^t_{ik}$ and $c^t_{ijkl}$, where $c^t_{ijkl}$ denotes the crossing variable that takes the value 1 if edges $(i, j)$ and $(k, l)$ cross. Furthermore, $x^t_{ik}$ takes the value 1 if vertex $i$ precedes vertex $k$ and 0 otherwise, where $i, k$ are vertices that reside in the same layer $t$, $t = 1, \ldots, p$. The mathematical formulation of the Graph Drawing Problem is as follows:

$$\text{(GDP)} \quad \min \quad \sum_{t=1}^{p-1} \sum_{(i,j),(k,l)\in E^t} c^t_{ijkl}$$

s.t.

$$-c^t_{ijkl} \le x^{t+1}_{jl} - x^t_{ik} \le c^t_{ijkl}, \qquad \forall\ (i,j),(k,l) \in E^t, j < l, \forall t \qquad (1)$$

$$1 - c^t_{ijkl} \le x^{t+1}_{lj} + x^t_{ik} \le 1 + c^t_{ijkl} \qquad \forall\ (i,j),(k,l) \in E^t, j > l, \forall t \qquad (2)$$

$$0 \le x^t_{ij} + x^t_{jk} - x^t_{ik} \le 1 \qquad \forall\ 1 \le i < j < k \le n_t, \forall t \qquad (3)$$

$$x^t_{ij}, c^t_{ijkl} \in \{0,1\}, \qquad \forall\ (i,j),(k,l) \in E^t, \forall t \qquad (4)$$

From an original hierarchical graph $H = (V, E, p, L)$ and its drawing $D = (H, \Pi_0)$, we can consider the addition of some vertices $\hat{V}$ and edges $\hat{E}$ obtaining an incremental graph $IH = (IV, IE, p, L)$, where $IV = V \cup \hat{V}$ and $IE = E \cup \hat{E}$, keeping the same number of layers $p$. As in the previous problem, the sets of vertices and edges can be written as sequences of disjoint sets $IV = IV^1 \cup \ldots \cup IV^p$ and $IE = IE^1 \cup \ldots \cup IE^{p-1}$, and denote $m_t$ the number of vertices in the incremental graph in layer $t$, i.e., $m_t = |IV^t|$. The Incremental Graph Drawing Problem (IGDP) consists of finding a drawing $ID = (IH, \Pi)$ that minimizes the number of edge crossings while keeping the same relative position between the original vertices as in the original drawing $D$. The mathematical programming formulation of IGDP is the same as GDP with new set of constraints that preserves the ordering of the original vertices. In mathematical terms, for each layer $t$ the following equations are added in the formulation GDP:

$$x^t_{ij} + x^t_{ji} = 1, \qquad \forall\ i,j \in V^t\ : 1 \le i < j \le m_t \qquad (5)$$

$$x^t_{ij} = 1, \qquad \forall\ i,j \in V^t\ :\ \pi^t_0(i) < \pi^t_0(j). \qquad (6)$$

Literature dealing with IGDP is scarce. In fact, we are only aware of paper [29], which is limited to bipartite graphs, recently extended in [39] to the multilayer case. In [29], the authors describe a branch-&-bound procedure that is tested on a relatively small graph and a meta-heuristic procedure based on GRASP [13] applied to medium- and large- sized instances. This seminal work is improved in [30] and extended to more than 2 layers in [39], where the authors propose a Variable Neighborhood Scatter Search for the IGDP.

In this paper, we propose an alternative approach in which, together with the relative position constraints, we also require that the positions of the original vertices are constrained to be close to their positions in the original drawing. The mathematical model for the Constrained Incremental Graph Drawing Problem (C-IGDP) is as follows:

$$\text{(C-IGDP)} \quad \min \quad C(D) = \sum_{t=1}^{p-1} \sum_{(i,j),(k,l)\in IE^t} c_{ijkl}^t$$

s.t.

$$-c_{ijkl}^t \leq x_{jl}^{t+1} - x_{ik}^t \leq c_{ijkl}^t, \qquad \forall\ (i,j),(k,l) \in IE^t, j < l,\ \forall t \tag{7}$$

$$1 - c_{ijkl}^t \leq x_{lj}^{t+1} + x_{ik}^t \leq 1 + c_{ijkl}^t, \qquad \forall\ (i,j),(k,l) \in IE^t, j > l\ \forall t \tag{8}$$

$$0 \leq x_{ij}^t + x_{jk}^t - x_{ik}^t \leq 1, \qquad \forall\ i,j,k \in IV^t, i < j < k,\ \forall t \tag{9}$$

$$x_{ij}^t + x_{ji}^t = 1, \qquad \forall\ 1 \leq i < j \leq m_t,\ \forall t \tag{10}$$

$$x_{ij}^t = 1, \qquad \forall\ i,j \in V^t,\ \pi_0^t(i) < \pi_0^t(j),\ \forall t \tag{11}$$

$$\max\{1, \pi_0^t(i) - K\} \leq \pi^t(i), \qquad \forall\ i \in V^t,\ \forall t \tag{12}$$

$$\min\{\pi_0^t(i) + K, m_t\} \geq \pi^t(i), \qquad \forall\ i \in V^t,\ \forall t \tag{13}$$

$$x_{ij}^t, c_{ijkl}^t \in \{0,1\}, \qquad \forall\ (i,j),(k,l) \in IE^t, \forall t. \tag{14}$$

Constraints (7)-(9) are straightforward adaptations of (1)-(3) in the (GDP) formulation. Constraints (10) and (11) preserve the ordering of the original vertices and are somehow equivalent to (5) and (6) in the incremental problem. Finally, new additional constraints, labeled in the formulation as (12) and (13), are required to restrict the position of the original vertices. Let $K$ be a parameter representing a distance slack between the original position and the new one. Without loss of generality, we suppose that $i$ is in layer $t$, and $\pi_0^t(i)$ is the original position of vertex $i$. The new position $\pi^t(i)$ where vertex $i$ can be relocated in the solution must be such that:

$$\max\{1, \pi_0^t(i) - K\} \leq \pi^t(i) \leq \min\{\pi_0^t(i) + K, m_t\},\ \forall\ i \in V, \tag{15}$$

where $m_t$ represents the number of vertices in the incremental graph in layer $t$. Table 2 summarizes all the nomenclature introduced in this section.

# 4 Solution Methods

In this section, we propose three GRASP constructions, `C1`, `C2`, `C3`, a memory-based construction, `C4`, and two improving methods: a local search and a tabu search. We consider their combination in four different algorithms for the C-IGDP problem:

- `GRASP1`: `C1` + Local Search.
- `GRASP2`: `C2` + Local Search.
- `GRASP3`: `C3` + Local Search.
- `TS`: `C4` + Tabu Search.

GRASP is a multi-start methodology that iteratively performs two phases, constructive and improvement, to achieve global diversification and local intensification in the search ([15]). The first phase employs a greedy randomized procedure to generate an initial solution, and then the second phase executes the local search to obtain a local optimum. The best solution

| Symbol | Definition |
|---|---|
| $G$ | Original graph: $G = (V, E)$ |
| $V$ | Set of original vertices of $G$ |
| $E$ | Set of original edges of $G$ |
| $p$ | Number of layers |
| $H$ | Hierarchical graph: $H = (V, E, p, L)$ |
| $IH$ | Incremental graph: $IH = (IV, IE, p, L)$ |
| $L$ | Function that indicates the label of the layer that contains each vertex |
| $\hat{V}$ | Set of incremental vertices |
| $IV$ | Set of vertices in the incremental graph $IH$: $IV = \hat{V} \cup V$ |
| $V^t$ | Set of vertices in layer $t$ in $H$: $V = V^1 \cup \ldots \cup V^p$ |
| $E^t$ | Set of edges from $V^t$ to $V^{t+1}$ in $H$: $E = E^1 \cup \ldots \cup E^{p-1}$ |
| $n_t$ | Number of original vertices in layer $t$: $n_t = \|V^t\|$ |
| $IV^t$ | Set of vertices in layer $t$ in $IH$: $IV = IV^1 \cup \ldots \cup IV^p$ |
| $m_t$ | Number of vertices in layer $t$: $m_t = \|IV^t\|$ |
| $IE$ | Set of edges in the incremental graph $IH$ |
| $D$ | Drawing: $D = (H, \Pi_0)$ |
| $\Pi_0$ | Set of permutation $\{\pi_0^1, \ldots, \pi_0^p\}$ |
| $ID$ | Incremental graph drawing: $ID = (IH, \Pi)$ |
| $\Pi$ | Set of permutation $\{\pi^1, \ldots, \pi^p\}$ |
| $C(D)$ | Number of crossings of a drawing $D$ |
| $\Lambda(v)$ | Set of all vertices adjacent to $v$ |
| $K$ | Constraint limit |

Table 2: Symbols and Definitions.

found over all iterations is returned as the result. On the other hand, memory construction methods are based on the strategy of applying recorded past information to generate new solutions ([19]). In particular, we consider in this section both constructive and improvement tabu search methods.

### 4.1 GRASP constructive methods

The construction phase in GRASP is iterative, greedy, and adaptive. Each initial solution is iteratively built by considering one element at a time. The addition of each element to the solution is guided by a greedy function and the elements are selected at random from a list of good candidates, the so-called Restricted Candidate List ($RCL$). We propose three different ways to obtain an initial solution, based on a greedy strategy. While applying these methods, we should keep in mind that each original vertex $v$ has to be placed in a position between $\max\{1, \pi_0(v) - K\}$ and $\min\{\pi_0(v) + K, m_{L(v)}\}$.

To design a constructive method, the greedy function selecting the element at each step is expected to be based on the objective function, and therefore the semi-greedy selection in GRASP has to reflect good values in terms of the objective. Note however, that if the objective function is relatively time-consuming to evaluate, it is common practice in heuristic search to employ an alternative evaluation to guide the method. This evaluation has to be fast and somehow connected with the objective value. In the context of graph drawing, the

objective function is the number of crossings and an alternative fast evaluation can be the vertex degree. On the hand, since we consider an incremental problem with two types of vertices, original and new, a constructive method could start either from a partial solution with the original vertices already placed or from scratch. To test these different strategies, we propose three constructive methods. The first one, `C1`, is based on the alternative evaluation based on the vertex degree and starts from scratch; the second one, `C2`, is also based on vertex degree but starts from a partial solution with the original vertices; finally, `C3` is based on the direct number of crossings and also starts from a partial solution.

Our first method, called `C1`, constructs an initial solution from scratch. The method starts with the random selection of a vertex $v$ among those with maximal degree. This vertex is placed in a random position in its layer, taking into account that if it is an original vertex, the position cannot be greater than $\min\{(\pi_0(v)+K, m_{L(v)}\}$ or less than $\max\{(1, \pi_0(v)-K)\}$. In the next steps, the candidate list $CL$ is formed by all the unassigned vertices, where the degree $\rho(v)$ of a vertex $v$ is calculated with respect to the partial solution. Elements of the restricted candidate list $RCL$ are all vertices $v$ whose degree $\rho(v)$ is within a percentage $\alpha \in [0,1]$ of the maximum degree $\rho_{max} = max\{\rho(v) : v \in CL\}$:

$$RCL = \{v \in CL : \rho(v) \geq \alpha\rho_{max}\}. \tag{16}$$

The next vertex $v^*$ to be added to the partial solution is randomly selected from $RCL$. The vertex $v^*$ is placed in its layer, say layer $l$, in the position prescribed by the barycenter method, $bc(v^*)$.

The barycenter is probably the most frequently applied method to order vertices in hierarchical graphs. It simply computes the average position of their neighbors and sorts vertices with respect to these numbers. More precisely, the barycenter method estimates the position of vertex $v^*$ in layer $l$ as the average position of its neighboring vertices, $\Lambda(v^*)$. In mathematical terms:

$$bc(v^*) = \frac{\sum_{u\in\Lambda_{l-1}(v^*)} \pi^{l-1}(u)}{2\,|\Lambda_{l-1}(v^*)|} + \frac{\sum_{u\in\Lambda_{l+1}(v^*)} \pi^{l+1}(u)}{2\,|\Lambda_{l+1}(v^*)|} \tag{17}$$

This vertex is placed in the closest feasible position prescribed by its barycenter, $bc(v^*)$ as in (17). This position is computed with respect to the adjacent vertices that are already in the partial solution. If the vertex is an original vertex, $v^* \in V$, a *feasible position* in the C-IGDP must satisfy the problem constraint and cannot be less than $\max\{1, \pi_0(v) - K\}$ or larger than $\min\{\pi_0(v) + K, m_{L(v)}\}$.

The second constructive method, called `C2`, considers that we already have a partial solution of the problem given by the original drawing $D = (G, \pi)$. As in [39], `C2` starts with $D$ and iteratively adds one incremental vertex in each iteration. Initially, the candidate list contains the incremental vertices $\hat{V} = IV \setminus V$. As in `C1`, the greedy function is based on the vertex's degree in the partial solution. The vertex $v^*$ to be included is selected at random from $RCL$, which contains all the unassigned incremental vertices with a degree higher than or equal to $\alpha$ times the maximum degree (16). The method computes the barycenter by means of equation (17) and inserts $v^*$ in the closest feasible position to it.

Figures 7 to 10 illustrate procedure `C2`. The incremental graph consists of three layers with four original vertices $0, 1, 2, 3$ and two incremental ones $A, B$ in the first layer, labeled Layer 0. Layer 1 has six vertices, three of them are original, $0, 1, 2$, and the other three are incremental, $A, B, C$. Finally, the last layer, labeled Layer 2, consists of six vertices, being

$0, 1, 2, 3$ the original ones, and $A, B$ the incremental ones. Figure 7 also shows all the edges between pairs of vertices. For example, vertex 0 in Layer 1 is connected to vertex 0 and $A$ in Layer 0, and to vertices 0 and $B$ in Layer 2. We consider in this example the position constraint value $K = 1$.

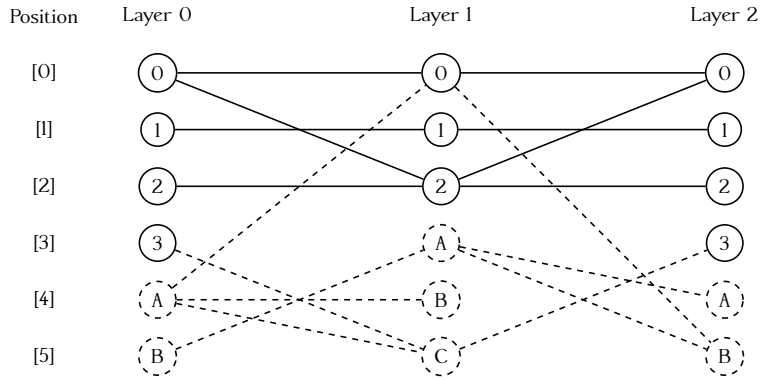


Figure 7: Graph to illustrate the `C2` method.

To feed the constructive method `C2`, all the original vertices are copied in the solution in consecutive positions. Figure 8 shows the initial partial solution with the original vertices in the first positions. Note that, positions [4] and [5] are free in Layer 0 and Layer 2 and in Layer 1 the free positions are [3], [4] and [5]. Firstly, $RCL$ elements are all the incremental vertices and then at each iteration, one of them is added to the partial solution.

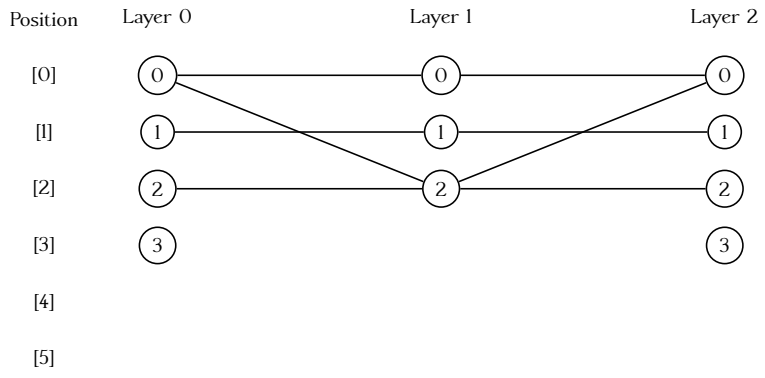


Figure 8: Initial partial solution.

Let us consider the partial solution in one iteration of the construction algorithm `C2`, shown in Figure 9. Four incremental vertices are already in the partial solution, vertex $A$ in Layer 0, vertices $B$ and $C$ in Layer 1 and vertex $B$ in Layer 2. In each layer, all original vertices were moved one position down in order to insert the incremental vertices, $A$ (Layer 0), $B$ (Layer 1) and $B$ (Layer 2) in the first position, called position [0]. These movements are feasible since $K$ is equal to 1, and each vertex can be moved one position up or down from its original position.

In this iteration the candidate list contains vertices $B$ from Layer 0, $A$ from Layer 1, and $A$ from Layer 2, too. Suppose that after the construction of the $RCL$ the vertex to be inserted in the partial solution is $A$ from Layer 1. If we compute its barycenter with equation

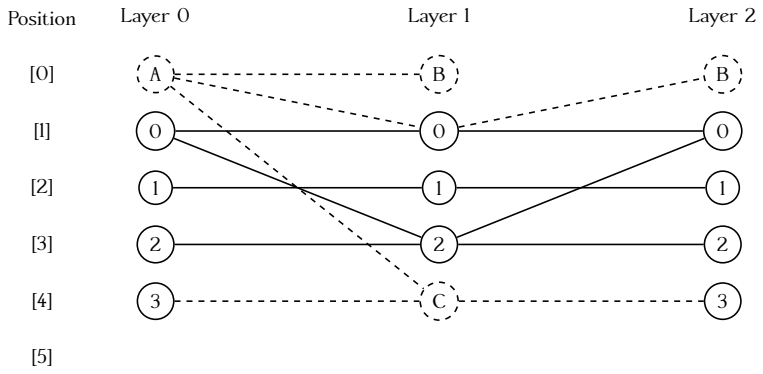


Figure 9: Partial solution in an iteration of `C2`.

(17), then $bc(A) = 0$. Note that to compute this value we only have to consider the vertices adjacent to $A$ that are already in the partial solution. However, in this particular case, $A$ has only one adjacent vertex in the solution, vertex $B$ (Layer 2).

As $bc(A) = 0$, the candidate position to insert $A$ in Layer 1 is [0], but this position is already occupied by vertex $B$. As the position constraint value is $K = 1$ and vertices 0, 1 and 2 were already shifted to one position down in previous iterations, then vertex $A$ cannot be inserted in positions [0], [1], [2] and [3]. We then insert $A$ in the closest feasible free position, which in this case is [4].

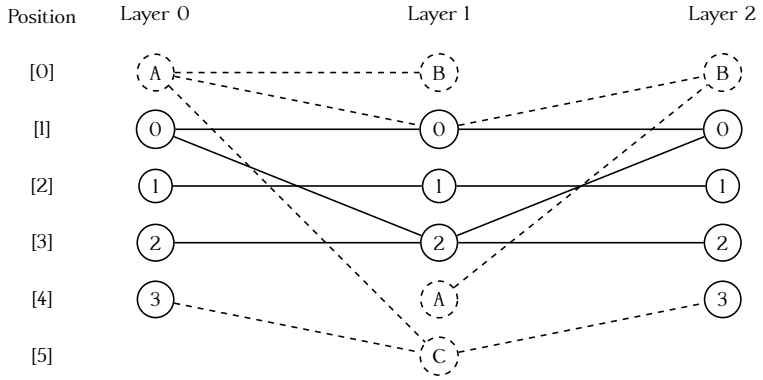


Figure 10: Partial solution after an iteration of `C2`.

In the basic GRASP construction phase, at each iteration, the choice of the next element to be added is determined by ordering all candidate elements, in a candidate list with respect to a myopic greedy function. This function measures the benefit of selecting each element. In the previous construction methods, `C1` and `C2`, this function is based on the vertices' degree. In our last construction method, `C3`, function $g(v, q)$ computes the number of edge crossings when vertex $v$ is inserted in position $q$ in its layer. To calculate the best insertion for vertex $v$, the greedy function $g(v)$ computes the minimum of the $g(v, q)$ values for all $q$-positions. In mathematical terms, $g(v) = \min_q g(v, q)$. If the vertex $v$ is selected, then it is placed in this best position in which the number of crossings is minimized. Initially, as in the case of `C2`, the method starts with the original drawing. In subsequent iterations, the candidate list $CL$ is formed with all unselected vertices and the $RCL$ consists of all vertices in $CL$ with the number

```
1  ID ← D;
2  CL ← IV \ V;
3  RCL ← ∅;
4  forall v ∈ CL do
5  |  compute g(v) and τ;
6  forall v ∈ CL do
7  |  if g(v) ≤ τ then
8  |  |  RCL ← RCL ∪ {v};
9  while ID is not complete do
10 |  v* ←select_node_randomly(RCL);
11 |  ID ← add_node_to_solution(v*);
12 |  recompute g and τ;
13 |  rebuild RCL;
14 return ID
```

Figure 11: Constructive phase `C3` for the C-IGDP.

of edge crossings lower than or equal to a threshold $\tau$. That is $RCL = \{v \in CL : g(v) \leq \tau\}$, where

$$\tau = \min_{v \in CL} g(v) + \alpha \left( \max_{v \in CL} g(v) - \min_{v \in CL} g(v) \right). \tag{18}$$

The parameter $\alpha$ controls the balance between the diversity and the quality of the solution, and it is empirically tuned (see Section 6). Then, a vertex $v^*$ is selected at random among the $RCL$ elements. If the position associated to $v^*$ is not free, then previous vertices are shifted up as in `C2` if it is possible. The pseudocode of construction phase `C3` is described in Figure 11. These three methods are compared in Section 6, and the best one is selected to be part of our solver for this problem. Note that we provide the reader with a detailed description of the three of them since we believe that these different strategies can be of interest when targeting other problems.

## 4.2 Memory construction procedure

GRASP constructions are memory-less methods, since no information is recorded and used from one construction to the next one. In other words, those techniques perform an independent sampling in the solution space. Conversely, it is maybe beneficial to save information from the past history thus designing constructions performing a guided selection in the solution space. We consider the inclusion of a frequency memory function $freq(\cdot, \cdot)$ to modify the evaluations of the greedy function with the inclusion of the recorded information. Specifically, we record in $freq(v, q)$ the number of times that vertex $v$ was inserted in the solution in position $q$ in previous iterations. Then, we modify the evaluation of the attractiveness of each non-selected vertex in the current construction to favor different types of solutions, which were not generated in previous iterations. Algorithm `C4` performs the same steps as `C3` but, instead of using the greedy function $g(v) = \min_q g(v, q)$, it uses:

$$g(v) = \min_q \left( g(v, q) + \beta freq(v, q) \right), \tag{19}$$

where $\beta$ is a critical parameter between $[0, 1]$. The first iteration of `C4` produces the same solution as `C3` since $freq(v, q) = 0$, for all vertices $v$ and positions $q$. But then, in subsequent iterations the method favors the selection of those vertices with low $freq(\cdot, \cdot)$ values.

### 4.3 Local Search Procedure

Our local search method explores each layer from 1 to $p$, one by one, searching for an improving move. We consider swapping the position of two incremental vertices as the first mechanism in the local search. We call $N_0(ID)$ this neighborhood where $ID$ is initially the solution obtained with one of the constructive methods described above. In a given layer, the method examines the incremental vertices, starting from the first one. For that vertex, we consider its possible swapping with all the incremental vertices in its layer.

```
1  best-cost ← C(ID);
2  ID* ← ID;
3  improvement ← true;
4  while improvement do
5  |  improvement ← false;
6  |  forall layers: t = 1...p do
7  |  |  forall vertices: v ∈ V^t | v is an incremental vertex do
8  |  |  |  best-swap ← −1;
9  |  |  |  forall vertices: v̄ ∈ V^t | v̄ is an incremental vertex do
10 |  |  |  |  if v̄ ≠ v then
11 |  |  |  |  |  ID* ← swap(v, v̄);
12 |  |  |  |  |  if C(ID*) < best-cost then
13 |  |  |  |  |  |  best-cost ← C(ID*);
14 |  |  |  |  |  |  best-swap ← v̄;
15 |  |  |  |  |  ID* ← swap(v̄, v);
16 |  |  |  |  if best-swap ≠ −1 then
17 |  |  |  |  |  ID* ← swap(v,best-swap);
18 |  |  |  |  |  improvement ← true;
19 return ID*;
```

Figure 12: `Swapping` phase in local search for C-IGDP.

The method performs the best feasible move if it improves the objective function (i.e., if it reduces the number of crossings). Then, it resorts to the next incremental vertex in the layer (following the current order) to try to swap it. Note that, since we only swap incremental vertices here, all these moves are feasible. When we finish the exploration of a layer, say $l$, we consider the next one, $l + 1$, and apply the same procedure. This local search performs multiple sweeps from the first to the last layer until no further improvement is possible. In short, it implements the so-called best strategy over a swapping move. We call *Swap* this local search phase based on neighborhood $N_0(ID)$, and its pseudo-code is reported in Figure 12.

We complement our local search with a second phase, called *Insertion*, based on a different neighborhood structure, $N_1(ID)$. Specifically, this phase scans the layers from 1 to $p$, and within each layer it considers the incremental vertices in their current ordering to perform

an insertion. Given an incremental vertex $v$, the method explores all its feasible insertions in previous position. Note that, unlike the former neighborhood structure, in this phase we have to check the feasibility of the original vertices. By a feasible move we mean that the position of original vertices is within its limits. If $\pi(v)$ is the position of vertex $v$ after the move, then $\pi(v) \in [\max(\pi_0(v) - K, 1), \min(\pi_0(v) + K, m_{L(v)})]$. If the best feasible move in a previous position improves the current solution, we perform it; otherwise we consider the insertions of $v$ in a posterior position, identifying the best feasible one. We perform the best move if it improves the solution. Once $v$ has been examined, we resort to the next incremental vertex (from $\hat{V}$) in this layer.

As in the previous phase, we perform sweeps from layer 1 to $p$ until no further improvement is possible. Our local search finishes when the two phases, *Swap* and *Insertion*, are performed, and returns the local optimum found.

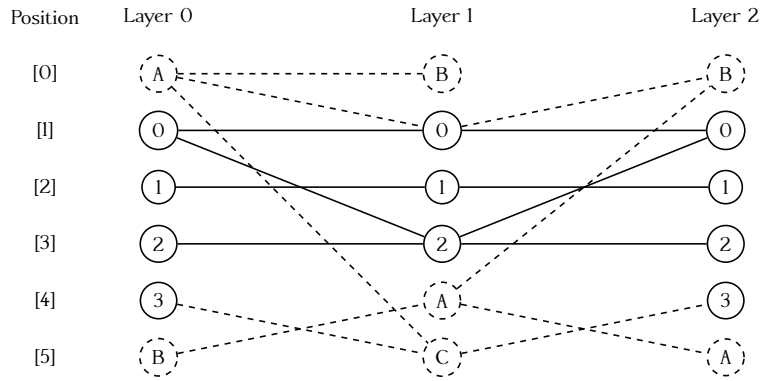


Figure 13: Initial solution.

Consider again the example in Section 4.1 and assume that after the construction phase shown in Figures 7 to 10, we obtain the feasible solution in Figure 13 with 13 crossings. In the first layer, Layer 0, swapping the vertices $A$ and $B$ does not decrease the number of crossings. The next layer to explore is Layer 1. In this case, exchanging positions of vertices $A$ and $C$ produces an improvement in the solution. The move is done and the number of crossings is reduced to 11, as shown in Figure 14-left. As in the first layer, the possible moves in last layer do not produce an improvement, then the first phase stops since no further swaps can reduce the number of crossings. In the second phase, in Layer 0 and 1 there are no insertion moves for the incremental vertices improving the quality of the solution. While, in Layer 2, by inserting the incremental vertex $B$ in position [1] the number of crossings is reduced from 11 to 10 (see Figure 14-right).

### 4.4 Tabu Search

To complement the memory based construction strategy, we implement a tabu search method, based on the neighborhood $N_1(ID)$. The procedure starts with the initial solution obtained by `C4` and operates as follows. It scans the layers from 1 to $p$, and for each layer, it evaluates all possible insertions of the new vertices and performs the best one, i.e., the one that produces the minimum number of crossings. It is worth mentioning that the tabu search method always performs the best available move, even if it is a non-improving move. The moved vertex is made tabu-active and remains tabu for *tenure* iterations, which means that

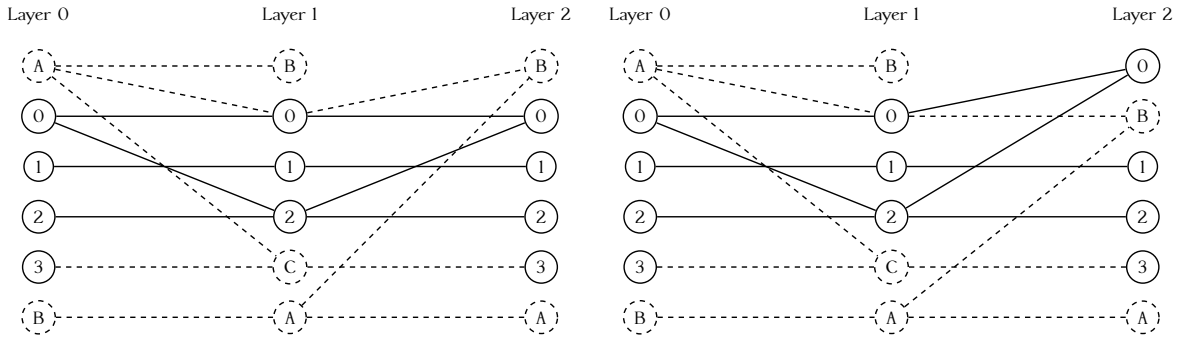


Figure 14: Local search procedure. Left: Swap phase. Right: Insertion phase.

during this period it cannot be moved. In the next steps, the insertions are only possible with non tabu-active vertices. As it is customary in the tabu search methodology, the tabu status is overridden if the movement leads to a solution that improves the best solution found so far. The tabu search method finishes when a pre-established number of iterations is met, and the best visited solution is returned as its output.

# 5 Path Relinking post-processing

Path Relinking (PR) is an approach suggested in the context of tabu search to combine solutions by creating paths between them [19]. GRASP hybridized with PR was first proposed by Laguna and Martí in [27] as an intensification method, and hybridizations of GRASP with Path Relinking are deeply discussed in [16]. We consider here the variant known as forward PR [39] for the C-IGDP.

Path Relinking is performed between two solutions, $ID^{ss}$ (source solution) and $ID^{ts}$ (target solution) for the C-IGDP. Each single move generated by the PR consists of replacing an entire layer from a current solution with a layer from the target solution. The total numbers of solutions generated by the PR during the path from $ID^{ss}$ to $ID^{ts}$ is $\frac{p(p+1)}{2} - 1$. For example, given the two solutions, $ID^{ss}$ and $ID^{ts}$ in Figure 15 with 3 layers and 3 vertices in each layer, the algorithm generates three different solutions exchanging one of the layers of $ID^{ss}$ with the layers of $ID^{ts}$. Then, the path continues from the intermediate solution $ID^2$, which is the best solution found, and generates another two solutions, $ID^4$ and $ID^5$. Finally, the path is completed from $ID^4$ to $ID^{ts}$. The method finds two solutions that improve the current one $ID^2$ and $ID^4$, both with number of crossings equal to 0.

The procedure operates on a set of solutions, called *Elite Set* ($ES$), constructed with the application of a previous method, in our case GRASP or Tabu Search. In order to obtain an elite set as good and diverse as possible, we start by considering as elite the $m = |ES|$ solutions obtained by our previous method. Then, in subsequent iterations, we analyze whether the generated solution $ID^*$ qualifies to enter in the elite set or not. In particular, if solution $ID^*$ is better than the best solution in $ES$, then the worst solution is replaced by $ID^*$. However, if $ID^*$ is better than the worst solution and it is sufficiently different (with a distance larger than a parameter $\gamma$) from the other elite solutions then it is also inserted in $ES$. In this latter case, it will replace the worst most similar solution. We define the diversity distance or the difference between two solutions as the number of positions that are not occupied by the same vertices divided by the number of vertices in the graph. The PR finishes when the

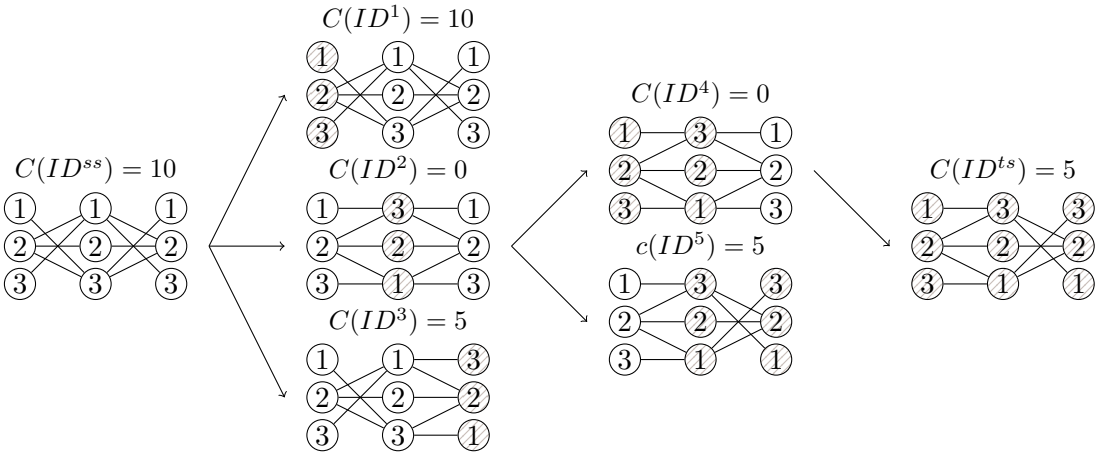

Figure 15: Example of the Path Relinking procedure.

paths between all pairs of elite solutions have been explored.

# 6 Computational Experiments

This section details our computational experiments to study the performance of the procedures presented above. In particular, we consider the following methods: `GRASP1` (`C1`+Local Search), `GRASP2` (`C2`+Local Search), `GRASP3` (`C3`+Local Search), and `TS` (`C4`+Tabu Search). Additionally, when we disclose the best GRASP variant, then we couple it with `PR`, calling the resulting method as `GRASP+PR`. In the following subsections, we first outline the experimental framework (see Subsection 6.1), and then we analyze the performance of these methods with respect to their parameters and settings in Subsection 6.2. This subsection also includes a comparison of the GRASP variants and Tabu Search. Finally, we compare in Subsection 6.3 our best heuristics with `LocalSolver`, and `CPLEX`, run with the integer linear programming model proposed above. `LocalSolver` is a well-known commercially available optimization software (localsolver.com). It is a black-box solver that uses metaheuristic methodologies to solve any combinatorial problem. It provides high-quality solutions in short computing times, and it is currently considered an alternative to customized heuristic methods.

## 6.1 Experimental Setup

All the procedures were implemented in C++, compiled with gcc 5.4.0, and the experiments were conducted on an Intel Corei7-4020HQ CPU @2.60Ghz x 8.

The set of instances used is available on-line in the web-page http://www.optsicom.es/igdp. In line with previous graph drawing papers [39], this set consists of 240 instances with four different numbers of layers 2, 6, 13 and 20, and 0.065, 0.175 and 0.3 graph densities. The number of layers $p$ is an input to the graph generator and the number of vertices in each layer is randomly chosen between 5 and 30. For each vertex u in layer $t$, an edge to a randomly chosen vertex v in layer $t + 1$ is included. In addition, the generator checks that all vertices in the last layer have a degree of at least one. If a vertex in layer $p$ is found with zero degree, an edge is added to a randomly chosen vertex in layer $p - 1$. The generator then adds enough edges to cover the difference between the current number and the number that results from the required density. We then applied the barycenter algorithm to obtain a drawing for

each graph. Finally, of the 20 instances generated for each combination of number of layers and density, 10 are augmented by incremented the number of vertices in 20% and 10 are augmented by 60%.

We divide the test set of 240 instances in four subsets according to the number of layers, $p = 2, 6, 13$, and 20, with 60 instances in each one. The average number of vertices per layer is approximately 25, and the average number of edges between consecutive layers is approximately 90. To give the reader an idea of the instances dimension, we can say that according to this average computation, an instance with for example 13 layers would have close to $n = 325$ vertices and 1080 edges. To avoid the over training of the methods, we consider a subset with a 10% of them, 24 instances, 6 in each subset, with different sizes and densities, to fine tune the parameters of the algorithms. We chose them to be representative of the entire set. Specifically, their number of vertices ranges from $n = 32$ (corresponding to a 2-layer instance) to $n = 960$ (a 20-layer instance). We call this subset the *training set*, as opposite to the entire set of instances called the *testing set*.

During our experiments, we report the following measures:

- $\bar{C}$: Average number of crossings.
- % *dev*: Average percent deviation with respect to the best solution found in the experiment.
- *Best*: Number of instances for which a procedure is able to match the best-known solution.
- *Score*: It is calculated as $(s(p_r - 1) - r)/(s(p_r - 1))$, where $p_r$ is the number of procedures being compared, $s$ is the number of instances, and $r$ is the number of instances in which the $p_r - 1$ competing procedures obtain a better result. Roughly speaking, the score is inversely proportional to the fraction of instances for which the competing procedures produce better solutions than the procedure being scored. Hence, the best score is 1 (when there is no better procedures, $r = 0$) and the worst is 0 (when all the procedures in each instances are better than the procedure being scored, $r = s(p_r - 1)$)
- $Time$: Time in seconds to execute the method.
- % *gap*: Average percentage deviation of the objective function value of the new method with respect to the solution value obtained with `CPLEX`.

An important element in the model is the value of the parameter $K$, which measures the distance between the original position of the vertices and the feasible ones. It is related to the problem definition since its objective is to keep vertices close to their original positions to preserve the user's mental map. It is clear that $K = 1$ satisfies this condition but the resulting problem is very constrained. We therefore consider low values of this parameter and test if a marginal increase would permit to obtain better solutions in terms of number of crossings. In particular, we test $K = 1, 2$, and 3. Therefore, for each instance in our set we generate three instances, one for each $K$ value in most cases (note that the number of added vertices in each layer has to be larger than $K$, thus some values of $K$ cannot be considered). Then, the training set has a total of 62 instances, and the testing set a total of 609 instances.

### 6.2 Preliminary Experiments

In this section, we first perform an experimental study to fine tune the algorithmic parameters of our methods. We have designed four constructive algorithms based on a greedy function (see Section 4.1). These constructive methods, namely `C1`, `C2`, `C3` and `C4` are parametrized by $\alpha$, which balances greediness and randomness. In this first experiment, we evaluate the influence of this parameter by considering three different values of $\alpha$: 0.25, 0.50, 0.75. We also include a variant labeled *random*, where the method randomly selects an $\alpha$ value in the range $[0, 1]$ for each construction. Table 3 shows the corresponding results when generating 500 independent constructions for each instance in the training set. This table shows for each procedure, the average number of crossings ($\bar{C}$), the average percent deviation from the best solution found (% *dev*), the number of best solutions (*Best*), the score statistic (*Score*) and the CPU-time in seconds required to execute the method (*Time*).

| $\alpha$ | $\overline{C}$ | % *dev* | *Best* | *Score* | *Time* |
|---|---|---|---|---|---|
| 0.25 | 17711.8 | 0.63 | 19 | 0.49 | 0.92 |
| 0.50 | 17747.1 | 0.69 | 12 | 0.47 | 0.96 |
| 0.75 | 17780.1 | 0.73 | 14 | 0.38 | 0.96 |
| *random* | **17685.7** | **0.16** | **39** | **0.87** | **0.93** |

Table 3: Fine-Tune parameter $\alpha$ for the constructive `C1` procedure.

Results in Table 3 clearly show that the best performance for `C1` is achieved by the *random* variant. This variant is able to obtain 39 best solutions out of the 62 instances, which compares favorably with the other variants. Likewise, we analyze the value of $\alpha$ for the constructive methods `C2`, `C3`, and `C4`, obtaining a similar result. Computational effort does not play a role here because all procedures build solutions in a negligible amount of time.

In our second preliminary experiment, we undertake to explore the memory construction procedure, `C4`, which has an additional search parameter, $\beta$, to be tuned. This parameter controls the number of times that a vertex has been selected in previous iterations. We test $\beta = 0.25$, 0.50, 0.75, and a random selection. Table 4 summarizes the results of this experiment with the same measures described above, and shows that the best solutions on average are obtained with $\beta$ selected at random in each iteration (*random* option).

| $\beta$ | $\overline{C}$ | % *dev* | *Best* | *Score* | *Time* |
|---|---|---|---|---|---|
| 0.25 | 17293.6 | 0.56 | 15 | 0.54 | 9.94 |
| 0.50 | 17288.1 | 0.82 | 9 | 0.43 | 8.83 |
| 0.75 | 17286.5 | 0.88 | 10 | 0.37 | 8.52 |
| *random* | **17248.1** | **0.30** | **33** | **0.73** | **9.27** |

Table 4: Fine-Tune parameter $\beta$ for the constructive `C4` procedure.

Table 5 compares the four construction procedures with their parameter values as indicated above in the training set instances. In order to run the methods for similar CPU times to perform a fair comparison, the number of constructions performed with each one is different. In particular, 1000 constructions for the `C1` and `C2` procedures, 100 for `C3` and `C4`. As in the

previous tables, the deviation values reported in this table are obtained considering the best solutions found in the experiment.

| **Procedure** | $\overline{C}$ | % *dev* | *Best* | *Score* | *Time* |
|---|---|---|---|---|---|
| `C1` | 17645.5 | 3.79 | 0 | 0.204 | 1.837 |
| `C2` | 17681.9 | 3.77 | 0 | 0.156 | 1.973 |
| `C3` | 17395.3 | 1.01 | 29 | 0.796 | 2.076 |
| `C4` | 17387.6 | 0.27 | 36 | 0.860 | 2.185 |

Table 5: Performance comparison of the construction methods.

Our local search method is based on two different moves, swaps and insertions. Initially, it performs swaps between incremental vertices, and then, in subsequent iterations, it implements insertion moves. In this preliminary experiment, we evaluate the effectiveness of each type of move (i.e., neighborhood structure) and its contribution to the final solution quality. In order to perform this analysis, we execute a `C3` constructive phase coupled with three different local search procedures. We also include the solutions of the constructive method with no local search as a baseline in the comparison (`C3`). The three methods are: swap-only neighborhood (`C3 + S`), insertion-only neighborhood (`C3 + I`), and both (`C3 + S + I`). These four setups are tested on the whole *training set*, for this experiments we keep track of: the average number of crossings, the mean percentage deviation from the best value, the number of best solution found, the score statistics and the time to best.

The results reported in Table 6 show that, as expected, the combination of the two neighborhood strategies achieves the maximum number of best solutions and best crossing average, with extremely low deviation and very high score statistic. For this reason, in the next experiments, the local search used for the GRASP algorithm consists of `S + I` as described in Section 4.3.

| **Procedure** | $\overline{C}$ | % *dev* | *Best* | *Score* | *Time* |
|---|---|---|---|---|---|
| `C3` | 17383.9 | 7.12 | 1 | 0.016 | 1.698 |
| `C3+S` | 16660.0 | 1.48 | 5 | 0.446 | 11.713 |
| `C3+I` | 16575.3 | 0.58 | 17 | 0.694 | 2.299 |
| `C3+S+I` | **16497.9** | **0.00** | **60** | **0.989** | 12.021 |

Table 6: Comparison among different local search setups.

An interesting question when comparing constructive methods is if they really need to produce high-quality solutions, or if their role is to obtain diverse initial solutions from which to apply the local search. To investigate this point, we perform an experiment to compare the value of the constructed solutions when applying `C3`, with the value of the improved ones after applying the local search method. In particular, we compute the correlation of these two values over 100 solutions for several instances. We obtain that in all the cases the correlation is relatively low. However, results are heterogeneous since in some instances the correlation is close to 0, while in others is close to 0.4. Figure 16-left shows the scatter-plot diagram of an instance with very low correlation, and Figure 16-right shows this diagram of another instance with a correlation close to 0.4. The $x$-axis represents the value of the constructed solution,

and the $y$-axis the value of the improved one. A point is plot for each pair of associated values. In both diagrams the points are scattered over the plane and they are not aligned over a straight line. Note, however, that they present different patterns in terms of their correlation.

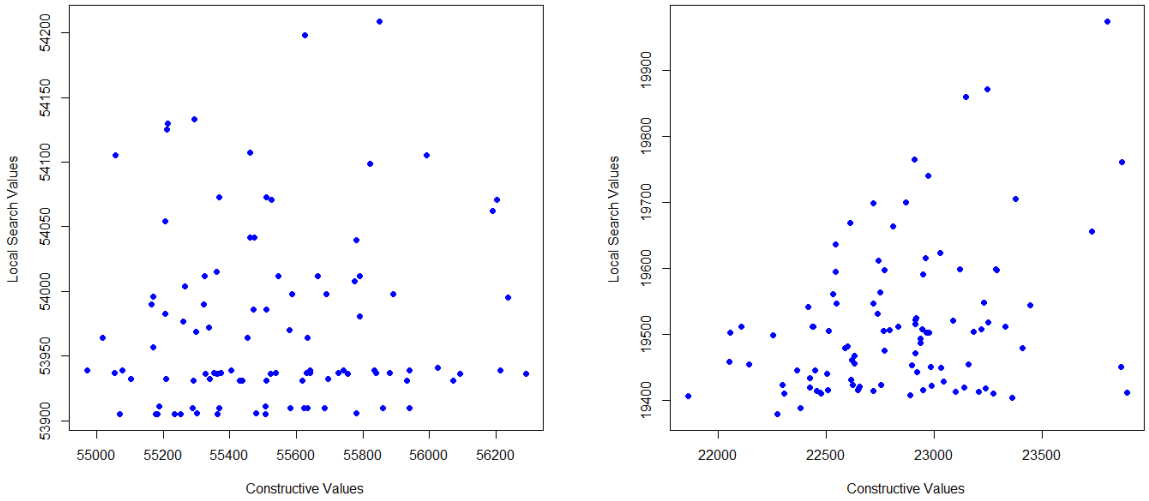


Figure 16: Scatter Plot for two different instances.

Since results in the previous experiment indicate a different pattern across the instances in terms of their correlation, we cannot conclude that we should construct good solutions to obtain good local optima. Additionally, diversity is relatively difficult to directly evaluate on the constructed solutions to test their ability to produce different local optima. Therefore, we compare the quality of the solutions obtained with the complete method, once the local search has been applied. Table 7 shows the results of the solutions when these constructive methods are coupled with the local search. We analyze the combination of the GRASP constructive methods `C1`, `C2`, and `C3` with the local search, and `C4` with the tabu search. They are denoted as `GRASP1`, `GRASP2`, `GRASP3`, and `TS` respectively. All of them generate and improve 100 solutions.

| **Procedure** | $\overline{C}$ | % *dev* | *Best* | *Score* | *Time* |
|---|---|---|---|---|---|
| `GRASP1` | 17193.7 | 0.14 | 307 | 0.63 | 7.61 |
| `GRASP2` | 17188.3 | 0.15 | 348 | 0.70 | 10.04 |
| `GRASP3` | 17188.3 | 0.13 | **358** | 0.72 | 11.12 |
| `TS` | 17183.3 | **0.04** | 335 | 0.73 | 34.36 |

Table 7: Performance comparison of GRASP variants and Tabu Search.

Table 7 shows that `GRASP3` is slightly better than `GRASP1` and `GRASP2`, both in average number of crossings ($\bar{C}$) and number of best results (*Best*). On the other hand, `TS` slightly outperforms `GRASP3` in terms of the average percentage deviation (% *dev*). However, it exhibits a lower number of best solutions (335, while `GRASP3` is able to obtain 358). We therefore

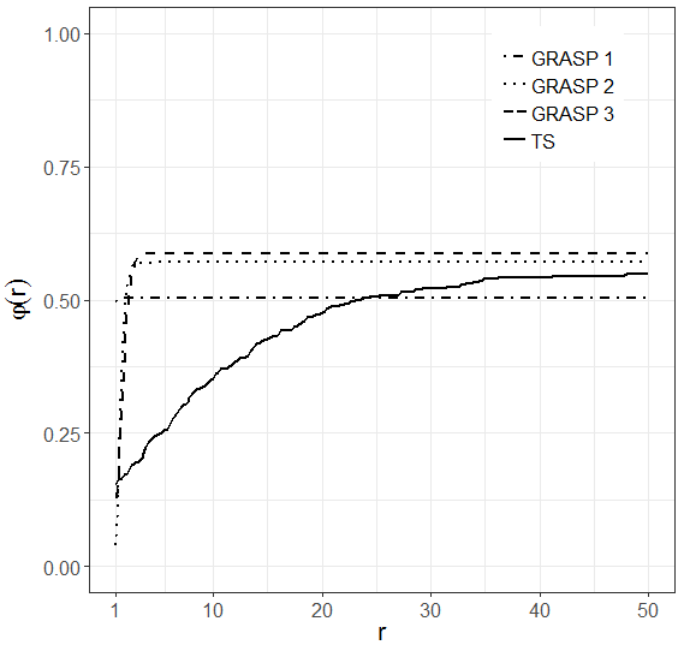

Figure 17: Performance profiles.

cannot conclude that one is better than the other, and select both `GRASP3` and `TS` as our solving methods to be applied in the rest of the experimentation.

We complement the comparative analysis of the findings with a performance profile plot, as described in [12]. This plot consists in the cumulative distribution function for a certain performance metric, which is the ratio of the computing time of each procedure versus the best time among all of them. The performance profile of each heuristic shows the probability $\varphi(r)$ that the ratio is within a factor $r \in \mathbb{R}$ of the best possible ratio. Figure 17 confirms how `GRASP1` is the fastest of the four heuristics. Indeed, this performance can be observed in $r = 1$, in which `GRASP1` shows the highest probability of achieving the best solution in the shortest time. This findings are consistent with the ones collected in Table 7, since the performance profile does not select multiple best solutions, but considers as best solution the one reached in the shortest time. At the same time, we can observe how considering higher computational efforts, the other two `GRASP` implementations are able to outperform `GRASP1`, with `GRASP3` exhibiting the highest probability to obtain the best solution. This switch in terms of performances happens for relatively low values of $r$, the upper bound on the relative time employed. Given the high number of best solution obtained and the quickly growing performance profile, we select `GRASP3` as the best-performing `GRASP` implementation. Moreover, the consistency evidenced by `TS` in Table 7, let us also consider `TS` as base methodology for our final experimentation.

Note that regarding the comparison between memory-less methods (GRASP in our case) and memory-based methods (Tabu Search), as we mentioned above, there is no clear winner. We cannot say that one systematically outperforms the other since each metric provides a different ranking. These results are in line with previous studies ([32], [9]), which also indicate that it is more a problem specific or implementation question, than a methodological one.

Our last preliminary experiment has the goal of setting the parameters of the Path Relinking post-processing. In particular, we set the value of the elite set size $|ES| = 3$, and the distance parameter $\gamma = 0.2$). We do not reproduce the table of this experiment, since as in

the previous ones, our selection is based on a trade-of between quality and computing time.

### 6.3 Comparison with Existing Methods

In this last section, we undertake to compare GRASP3 and TS methods when solving the C-IGDP, as well as to compare them with other previous methods: CPLEX and LocalSolver. Additionally, we evaluate the contribution of the Path Relinking post-processing to the final quality of the solution. Figure 18 shows the typical search profile in which the current solution value is represented at different times in the search process. Specifically, this figure shows the time in seconds on the $x-$axis, and the objective function value on the $y-$axis of an instance with 444 vertices (a 20% of them are incremental) and 2228 edges.

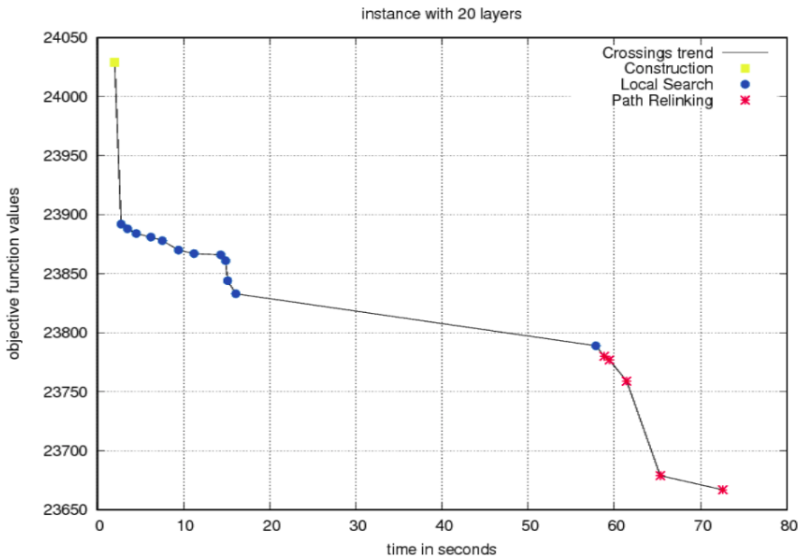


Figure 18: GRASP3+PR Search Profile.

Figure 18 shows the value of the solution constructed with C3 at 2 seconds. Then, when we move to the right-hand-side of the diagram, from 2 seconds to almost 60 seconds, we can see how the local search in GRASP3 is able to improve the solution from a value close to 24,000 to a value of 23,800. In the final stage of the profile, after 60 seconds, we can easily identify the PR application since the search profile shows a significant improvement in the value, which ends close to 23,650. Although this diagram only shows the performance for a single instance with 20 layers, we have empirically found that this is a typical performance for different instances.

Figure 18 shows the contribution of PR to the final solution of our complete method GRASP3+PR. We now complement this analysis by comparing GRASP3 with and without PR. Specifically, Figure 19 shows the search profile of GRASP3 over 150 seconds, and also the profile of a method consisting of applying first GRASP3 for 100 seconds, and then PR for the remaining 50 seconds. This figure clearly shows that it is worth investing the final search time on PR instead of continuously applying GRASP3 for the entire search. Note that at a certain time of the search GRASP3 is not able to further improve the solution and the profile stagnates on a certain value. At that point (around 100 seconds), PR is able to further improve the current solution. We can conclude that GRASP3+PR obtains high quality solutions, better than GRASP3

over a relatively long-term horizon.

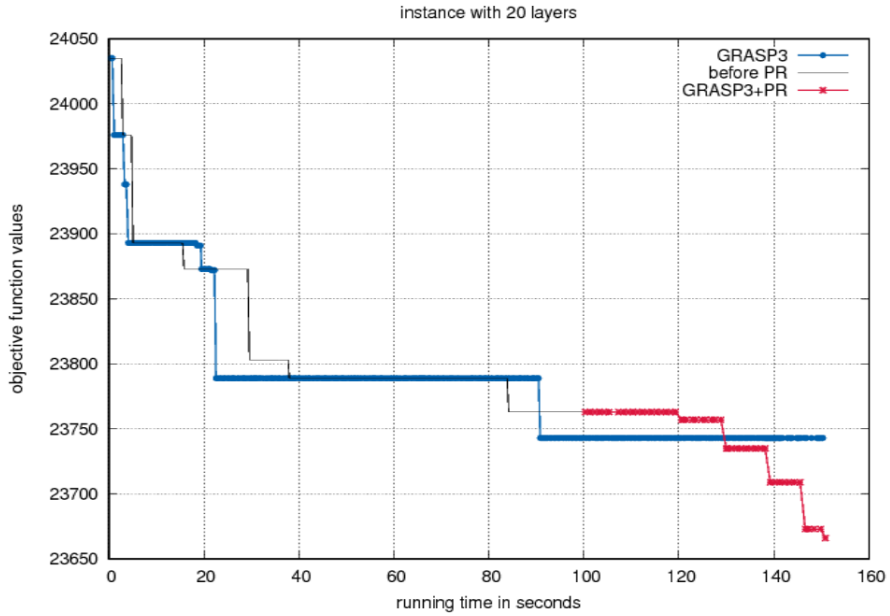


Figure 19: Search Profile.

We now compare `GRASP3, TS`, `GRASP3+PR`, and `TS+PR` with `CPLEX` and `LocalSolver` over the entire set of 609 instances. Table 8 shows the associated results classified by size. In this experiment, we run our heuristics for 100 iterations, which corresponds to moderate running times (from 0.5 to 50 seconds depending on the instance size). Since `CPLEX` performs an implicit enumeration from the mathematical model proposed in Section 3, it requires larger running times, so we configure it to run for a maximum of $1,800$ seconds. To give `LocalSolver` the opportunity to reach high-quality solutions, it is run with a time limit of 20 seconds on the instances with 2 layers, and $60, 150$ and 300 seconds on those with $6, 13$ and 20 layers respectively. As in previous experiments, this table shows, for each procedure, the average of number of crossings, the average percentage deviation from the best solution found, the number of best solutions, and the CPU-time in seconds. In addition, we also show the average percentage deviation between the heuristic solution value and the `CPLEX` best solution value ($\%gap$). Note that `CPLEX` is able to obtain the optimal solution in 572 out of the 609 instances.

Table 8 shows that, as expected, `GRASP3+PR` consistently obtains better results than `GRASP3`, and similarly `TS+PR` improves upon `TS`. Note that in the `PR` variants, only a small fraction of the total time is employed by `PR`, since it has the role of a post-processing method. It is worth mentioning that `CPLEX` is able to obtain the exact solutions for the small instances in similar time than our heuristic methods. However, the situation changes when we move to the large instances, where heuristics are able to obtain solutions of similar quality than `CPLEX` in lower running times. In particular, we can see that in the instances with 20 layers, `GRASP3+PR` and `TS+PR` have an average gap value of $-0.12$ and $-0.19$ respectively, which indicates that the heuristic outperforms `CPLEX` on average. Note that this table also indicates that `CPLEX` obtains the optimal solution in 116 instances with 20 layers, while our heuristics only obtain a fraction of these optimal solutions. However, this is achieved by `CPLEX` at a large

| **Procedures** | $\overline{C}$ | % *gap* | % *dev* | *Bests* | *Opt* | *Time* |
|---|---|---|---|---|---|---|
| 2 Layers ($32 \leq n \leq 96$), 171 instances | | | | | | |
| CPLEX | 2408.50 | - | 0.00 | 171 | 171 | 0.77 |
| GRASP3 | 2409.19 | 0.14 | 0.14 | 161 | 161 | 1.11 |
| GRASP3+PR | 2408.92 | 0.14 | 0.14 | 164 | 164 | 1.18 |
| TS | 2408.58 | 0.00 | 0.00 | 163 | 163 | 1.03 |
| TS+PR | 2408.51 | 0.00 | 0.00 | 170 | 170 | 2.73 |
| LocalSolver | 2785.47 | 17.01 | 17.01 | 12 | 12 | 20.11 |
| 6 Layers ($48 \leq n \leq 288$), 159 instances | | | | | | |
| CPLEX | 9995.70 | - | 0.01 | 157 | 157 | 56.24 |
| GRASP3 | 9997.43 | 0.13 | 0.14 | 81 | 81 | 5.53 |
| GRASP3+PR | 9994.32 | 0.08 | 0.08 | 100 | 98 | 5.39 |
| TS | 9999.73 | 0.20 | 0.20 | 63 | 63 | 5.48 |
| TS+PR | 9994.19 | 0.05 | 0.06 | 93 | 93 | 16.70 |
| LocalSolver | 11024.89 | 16.59 | 16.6 | 0 | 0 | 62.43 |
| 13 Layers ($104 \leq n \leq 611$), 141 instances | | | | | | |
| CPLEX | 23469.05 | - | 0.21 | 129 | 128 | 273.77 |
| GRASP3 | 23319.41 | 0.13 | 0.33 | 28 | 27 | 15.22 |
| GRASP3+PR | 23305.22 | 0.02 | 0.22 | 49 | 44 | 15.34 |
| TS | 23334.38 | 0.24 | 0.44 | 26 | 26 | 15.64 |
| TS+PR | 23301.16 | -0.07 | 0.14 | 38 | 31 | 47.42 |
| LocalSolver | 25530.24 | 15.95 | 16.16 | 0 | 0 | 162.06 |
| 20 Layers ($120 \leq n \leq 960$), 138 instances | | | | | | |
| CPLEX | 37918.20 | - | 0.38 | 118 | 116 | 383.73 |
| GRASP3 | 37522.42 | 0.03 | 0.39 | 21 | 20 | 25.77 |
| GRASP3+PR | 37495.44 | -0.12 | 0.24 | 36 | 29 | 28.39 |
| TS | 37540.99 | 0.11 | 0.47 | 22 | 22 | 26.72 |
| TS+PR | 37486.65 | -0.19 | 0.17 | 39 | 26 | 86.26 |
| LocalSolver | 40954.07 | 14.30 | 14.68 | 0 | 0 | 328.77 |

Table 8: Comparison on entire benchmark set according to instance size

computational cost, and additionally, for the remaining instances in this set where CPLEX is unable to obtain the optima, GRASP3+PR and TS+PR obtain better solutions, which cause the average gap to take negative numbers. Note that our heuristics only employ a fraction of the running time required by CPLEX, which in many cases employs the total time permitted of $1,800$ seconds, as evidenced by the long average running times shown in this table. Table 8 also shows that LocalSolver presents a poor performance since in spite of running it for longer CPU times than the competing heuristics, it exhibits the largest deviations with respect to the best known solutions and optimal values.

We now summarize the results in this experiment in a different way. In particular, Table 9 shows the average value of the statistics considered aggregating the instances by their $K$-value. We can observe more differences between the heuristics gap and deviations with $K = 3$

than with $K = 1$, with this being due to the fact that CPLEX is able to solve almost every instance with $K = 1$. In line with the analysis in Table 8, we can conclude that our heuristics are performing well when comparing them with CPLEX and LocalSolver.

| **Procedures** | $\overline{C}$ | % *gap* | % *dev* | *Bests* | *Opt* | *Time* |
|---|---|---|---|---|---|---|
| $K = 1$, 240 instances | | | | | | |
| CPLEX | 17196.63 | - | 0.03 | 238 | 238 | 49.85 |
| GRASP3 | 17164.45 | 0.03 | 0.06 | 132 | 131 | 5.89 |
| GRASP3+PR | 17160.88 | 0.00 | 0.04 | 157 | 156 | 6.32 |
| TS | 17162.75 | 0.02 | 0.05 | 149 | 149 | 6.27 |
| TS+PR | 17159.60 | 0.00 | 0.03 | 159 | 158 | 23.92 |
| LocalSolver | 18205.27 | 9.45 | 9.49 | 6 | 6 | 141.80 |
| $K = 2$, 210 instances | | | | | | |
| CPLEX | 17323.09 | - | 0.15 | 195 | 194 | 184.94 |
| GRASP3 | 17183.44 | 0.06 | 0.20 | 93 | 93 | 11.01 |
| GRASP3+PR | 17172.75 | -0.02 | 0.13 | 115 | 109 | 11.78 |
| TS | 17190.76 | 0.11 | 0.26 | 75 | 75 | 11.39 |
| TS+PR | 17170.58 | -0.06 | 0.08 | 104 | 95 | 36.29 |
| LocalSolver | 19164.93 | 17.11 | 17.26 | 4 | 4 | 134.46 |
| $K = 3$, 159 instances | | | | | | |
| CPLEX | 17471.57 | - | 0.27 | 142 | 140 | 313.39 |
| GRASP3 | 17230.84 | 0.30 | 0.56 | 66 | 65 | 19.17 |
| GRASP3+PR | 17210.94 | 0.15 | 0.41 | 77 | 70 | 19.80 |
| TS | 17254.76 | 0.33 | 0.58 | 50 | 50 | 19.15 |
| TS+PR | 17203.96 | -0.08 | 0.17 | 77 | 67 | 52.53 |
| LocalSolver | 19413.82 | 24.57 | 24.85 | 2 | 2 | 121.49 |

Table 9: Comparison on entire benchmark set according to $K$ value

We complement our comparison with 10 additional very large instances with 50 layers and $1,000$ vertices. These instances are too large to be solved with CPLEX, therefore we limit this comparison to heuristic methods. Additionally, considering that graph drawing systems require fast methods to provide the user with good solutions in very short computing time, we select GRASP3+PR as our best method for its trade-off between solution quality and running time. We therefore limit this final comparison to GRASP3+PR and LocalSolver. To set up a benchmark for future comparison, we report in Table 10 with $K = 1$, Table 11 with $K = 2$, and Table 12 with $K = 3$, the individual results of both methods on each instance. They run for 100 iterations for GRASP3+PR and a time limit of 900 seconds for the LocalSolver on each instance. Results in these tables clearly show the superiority of our procedure with respect to Local Solver in both solution quality and speed. Additionally, when comparing the solution value for different values of $K$, we can conclude that only a marginal improvement in the number of crossing is achieved when $K$ increases, and therefore low values of $K$ are recommended for the sake of stability in the sequence of drawings.

We perform now the so-called time to target plot, see [1], for this large instance with 20

| Instance | GRASP3+PR Crossings | *Time* | LocalSolver Crossings | *Time* |
|---|---|---|---|---|
| L50N1000.0 | 53048 | 49.92 | 59220 | 1019.69 |
| L50N1000.1 | 47051 | 42.85 | 52037 | 1003.95 |
| L50N1000.2 | 72018 | 64.73 | 78800 | 1058.08 |
| L50N1000.3 | 101428 | 93.84 | 110649 | 1127.93 |
| L50N1000.4 | 55112 | 51.73 | 61279 | 1016.93 |
| L50N1000.5 | 89599 | 76.68 | 97372 | 1093.13 |
| L50N1000.6 | 58829 | 54.32 | 65527 | 1023.40 |
| L50N1000.7 | 79236 | 75.52 | 87359 | 1069.29 |
| L50N1000.8 | 58147 | 55.45 | 65320 | 1028.31 |
| L50N1000.9 | 84496 | 79.64 | 92960 | 1112.17 |

Table 10: Best values on very large instances, with $K = 1$.

| Instance | GRASP3+PR Crossings | *Time* | LocalSolver Crossings | *Time* |
|---|---|---|---|---|
| L50N1000.0 | 52010 | 77.02 | 62547 | 1020.65 |
| L50N1000.1 | 46152 | 66.63 | 54934 | 1004.17 |
| L50N1000.2 | 70711 | 99.65 | 83344 | 1058.18 |
| L50N1000.3 | 99767 | 142.35 | 108847 | 1128.66 |
| L50N1000.4 | 54020 | 79.18 | 63984 | 1016.26 |
| L50N1000.5 | 88247 | 115.74 | 100745 | 1093.07 |
| L50N1000.6 | 57639 | 84.00 | 67388 | 1023.72 |
| L50N1000.7 | 77716 | 111.58 | 91685 | 1070.21 |
| L50N1000.8 | 57160 | 83.56 | 68692 | 1028.18 |
| L50N1000.9 | 82910 | 123.70 | 96252 | 1113.10 |

Table 11: Best values on very large instances, with $K = 2$.

layers and 512 vertices. This diagram shows the ability of a heuristic to match a target value (optimal value in our case). In particular, we run our heuristic GRASP3 for $n = 100$ trials and record the time to reach that target in each run. We sort the time values in increasing order: $t_1, \ldots, t_n$. The time to target plot, showed in Figure 20, depicts the cumulative probability $p_i = (i - 1/2)/n$ for each time value $t_i$ for $i = 1, \ldots, n$. The plots in this figure show the expected exponential runtime distribution for GRASP3. Therefore, linear speed is expected if the algorithm is implemented in parallel.

To finish our experimentation we consider the example shown in Figure 5 and obtain the incremental drawing with our new GRASP3+PR. Figure 21 shows the output of our method with $K = 4$. It is easy to check that the vertices in this figure are close to their original position in Figure 5. For example, vertices 3 and 12, which were in positions 1 and 17 respectively, are now in positions 2 and 21. The number of edge crossings of this graph is 4961, which compares favorably with the 6963 crossings of the initial drawing. Note, however, that the solution in Figure 6 has 4647 crossings. This is expected since we are solving a more constrained model. We believe that the marginal increase in the number of crossings of

| Instance | GRASP3+PR Crossings | *Time* | LocalSolver Crossings | *Time* |
|---|---|---|---|---|
| L50N1000.0 | 51413 | 99.90 | 63430 | 1019.81 |
| L50N1000.1 | 45361 | 83.13 | 57458 | 1004.67 |
| L50N1000.2 | 69940 | 132.82 | 84770 | 1058.01 |
| L50N1000.3 | 98704 | 187.58 | 120328 | 1128.18 |
| L50N1000.4 | 53298 | 106.53 | 65782 | 1016.16 |
| L50N1000.5 | 87324 | 155.52 | 103617 | 1093.30 |
| L50N1000.6 | 56783 | 111.33 | 70991 | 1024.39 |
| L50N1000.7 | 76828 | 152.64 | 94363 | 1072.14 |
| L50N1000.8 | 56527 | 106.78 | 70587 | 1027.96 |
| L50N1000.9 | 81838 | 160.89 | 91291 | 1112.31 |

Table 12: Best values on very large instances, with $K = 3$.

our model is completely justified by the stability obtained in the incremental drawing. In other words, the experimentation shows that our proposal provides a good trade-off between crossing reduction and drawing stability.

# 7 Conclusions

We have developed a heuristic procedure based on the GRASP methodology to provide high quality solutions to the problem of minimizing straight-line crossings in hierarchical graphs with an additional constraint. This problem is known as incremental graph drawing and the additional constraint models the stability on a sequence of drawings (the so-called user's mental map) when some vertices and edges are added by means of a parameter $K$. Our method is coupled with a Path Relinking post-processing to obtain improved solutions in the long term. We also tested a tabu search procedure to evaluate the contribution of memory structures in comparison with semi-random designs. Exhaustive experimentation first discloses the best configuration of our methods and then performs an empirical comparison with the existing ones, namely the general purpose solvers `CPLEX` and `LocalSolver`. Our `GRASP` and `TS` implementations were shown to be competitive in a set of problem instances for which the optimal solutions are known, and clearly outperform `LocalSolver`. Finally, as revealed on large instances, the larger the parameter $K$ the lower the number of crossings. However, this improvement is just marginal and therefore low values of $K$ (close to 1) are recommended to obtain good stable graphs.

As mentioned in the introduction, one of the objectives of this study is to disclose if memory structures are a better way to achieve diversification in search methods than semi-random designs. Our experimentation reveals that random elements present a marginal benefit with respect to memory elements in this problem, although different metrics when analyzing the results lead to *different winners*. In short, we can conclude that they present a similar performance. This conclusion is in line with previous studies, in which depending on the problem, one design may perform slightly better than the other. For example, in the context of clustering problems [32] iterated greedy, a well-known memory-based metaheuristic, performs slightly better than GRASP, while on the other hand, when maximizing diversity in location

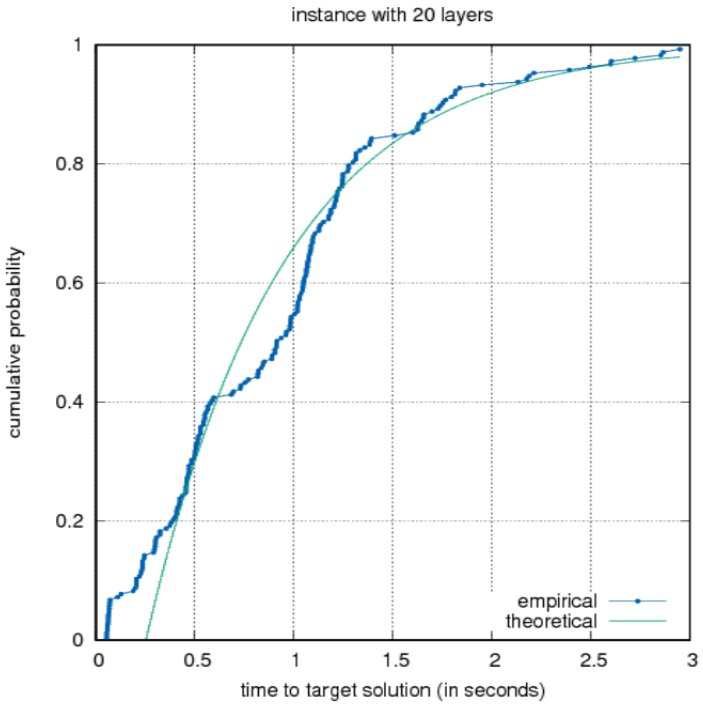


Figure 20: Time to target plot.

problems, it is well documented that memory based designs in general, and tabu search in particular, perform better than GRASP [9].

**Acknowledgments.** This work has been partially supported by the Spanish Ministerio de Economía y Competitividad with grant ref. TIN2015-65460-C02.

## References

[1] R. M. Aiex, M. G. C. Resende, and C. C. Ribeiro. Ttt plots: a perl program to create time-to-target plots. *Optimization Letters*, 1(4):355–366, Sep 2007.

[2] I. Antonellis, H. G. Molina, and C. Chao. Simrank++: query rewriting through link analysis of the click graph. *Proceedings of the VLDB Endowmen*, 1(1):408–421, 2008.

[3] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1998.

[4] S. Bhatt and F. Leighton. A framework for solving vlsi graph layout problems. *Journal of Computer and System Sciences*, 28:300–343, 1984.

[5] K-F. Böhringer and F. N. Paulisch. Using constraints to achieve stability in automatic graph layout algorithms. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '90, pages 43–51, New York, NY, USA, 1990. ACM.

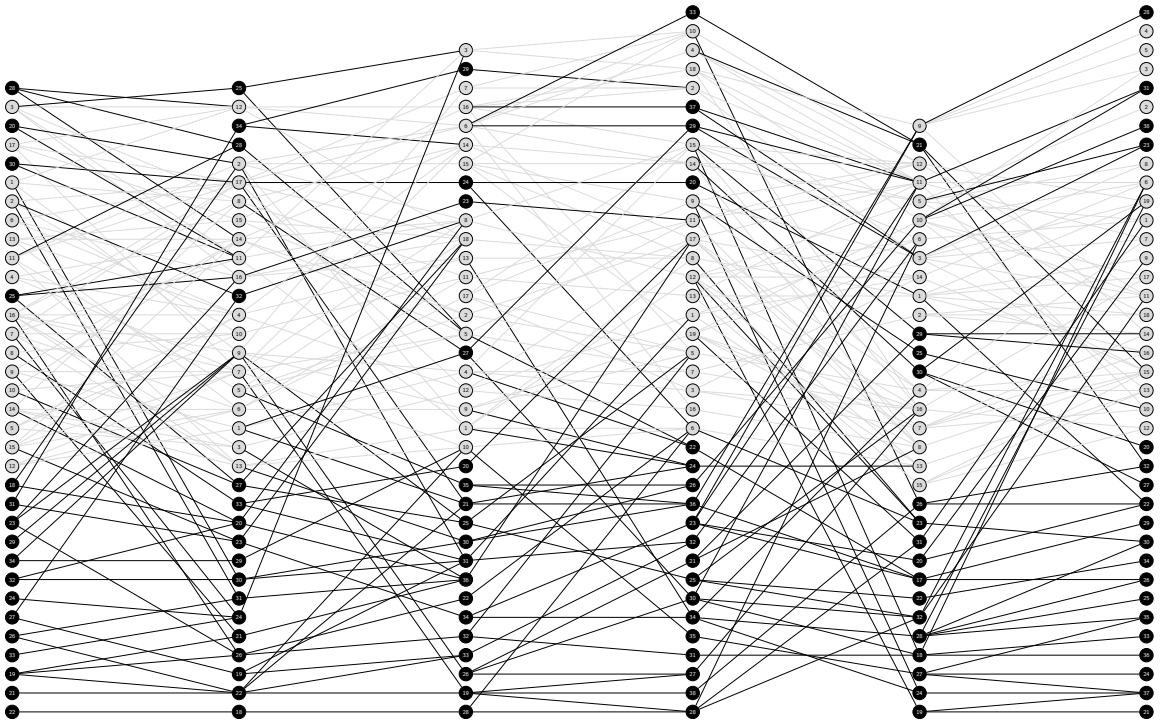

Figure 21: Output of `GRASP3+PR`.


[6] J. Branke. *Dynamic Graph Drawing*, pages 228–246. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[7] S. Bridgeman and R. Tamassia. A user study in similarity measures for graph drawing. *J. Graph Algorithms Appl.*, 6(3):225–254, 2002.

[8] M. Burch, C. Müller, G. Reina, H. Schmauder, M. Greis, and D. Weiskopf. Visualizing Dynamic Call Graphs. In M. Goesele, T. Grosch, H. Theisel, K. Toennies, and B. Preim, editors, *Vision, Modeling and Visualization*. The Eurographics Association, 2012.

[9] R. Carrasco, A. Pham, M. Gallego, F. Gortázar, R. Martí, and A. Duarte. Tabu search for the max-mean dispersion problem. *Knowledge based systems*, 85:256–264.

[10] J. Chen and I. T. Chau. The hierarchical dependence diagram: improving design for reuse in object-oriented software development. In *Proceedings of 1996 Australian Software Engineering Conference*, pages 155–166, Jul 1996.

[11] S. Diehl and C. Görg. Graphs, they are changing. In *10th International Symposium on Graph Drawing GD 2002*, page 23–30. Springer, 2002.

[12] Elizabeth D Dolan and Jorge J Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91(2):201–213, 2002.

[13] T. A. Feo and M. G. C. Resende. Greedy randomized adaptive search procedures. *Journal of global optimization*, 6(2):109–133, 1995.

[14] M. Fernández-Ropero, R. Pérez-Castillo, and M. Piattini. Graph-based business process model refactoring.

[15] P. Festa and M. G. C. Resende. An annotated bibliography of grasp–part i: Algorithms. *International Transactions in Operational Research*, 16(1):1–24, 2009.

[16] P. Festa and M.G.C. Resende. *Hybridizations of GRASP with Path-Relinking*, volume 434, pages 135–155. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[17] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000.

[18] M. R. Garey and D. S. Johnson. Crossing number is np-complete. *SIAM Journal on Algebraic and Discrete Methods*, 4:312–316, 1983.

[19] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[20] T. Gschwind, J. Pinggera, S. Zugal, H. A. Reijers, and B. Weber. A linear time layout algorithm for business process models. *J. Vis. Lang. Comput.*, 25(2):117–132, April 2014.

[21] C. Hu, Y. Li, X. Cheng, and Z. Liu. A virtual dataspaces model for large-scale materials scientific data access. *Future Generation Computer Systems*, 54:456 – 468, 2016.

[22] M. Jünger and P. Mutzel. 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. *Journal of Graph Algorithms and Applications*, 1:Paper 1, 25 p.–Paper 1, 25 p., 1997.

[23] M. Jünger, E. K. Lee, P. Mutzel, and T. Odenthal. A polyhedral approach to the multilayer crossing minimization problem. In *International Symposium on Graph Drawing*, pages 13–24. Springer, 1997.

[24] M. Kaufmann and D. Wagner, editors. *Drawing Graphs: Methods and Models*. Springer-Verlag, London, UK, UK, 2001.

[25] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[26] H.-P. Kriegel, P. Kröger, M. Renz, and T. Schmidt. Hierarchical graph embedding for efficient query processing in very large traffic networks. In B. Ludäscher and N. Mamoulis, editors, *Scientific and Statistical Database Management*, pages 150–167, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[27] M. Laguna and R. Martí. Grasp and path relinking for 2-layer straight line crossing minimization. *INFORMS Journal on Computing*, 11:44–52, 1999.

[28] N. W. Lemons, B. Hu, and W. S. Hlavacek. Hierarchical graphs for rule-based modeling of biochemical systems. *BMC Bioinformatics*, 12(1):45, Feb 2011.

[29] R. Martí and V. Estruch. Incremental bipartite drawing problem. *Computers & Operations Research*, 28(13):1287–1298, 2001.

[30] R. Martí, A. Martínez-Gavara, J. Sánchez-Oro, and A. Duarte. Tabu search for the dynamic bipartite drawing problem. *Computers & Operations Research*, 91:1–12, 2018.

[31] A. Martínez-Gavara, V. Campos, M. Gallego, M. Laguna, and R. Martí. Tabu search and grasp for the capacitated clustering problem. *Computational Optimization and Applications*, 62(2):589–607, Nov 2015.

[32] A. Martínez-Gavara, D. Landa-Silva, V. Campos, and R. Martí. Randomized heuristics for the capacitated clustering problem. *Information Sciences*, 417:154–168.

[33] C. Matuszewski, R. Schönfeld, and P. Molitor. Using sifting for k-layer straightline crossing minimization. In J. Kratochvíyl, editor, *Graph Drawing*, pages 217–224, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[34] S. C. North. Incremental layout in dynadag. In *In Proceedings of the 4th Symposium on Graph Drawing (GD*, pages 409–418. Springer-Verlag, 1996.

[35] B. Oselio, A. Kulesza, and A. O. Hero. Multi-layer graph analytics for social networks. In *2013 5th IEEE International Workshop on Computational Advances in Multi-Sensor Adaptive Processing (CAMSAP)*, pages 284–287, Dec 2013.

[36] B. Pinaud, P. Kuntz, and R. Lehn. *Dynamic Graph Drawing with a Hybridized Genetic Algorithm*, pages 365–375. Springer London, London, 2004.

[37] R. Marinescu R. Mateescu, R. Dechter. And/or multi-valued decision diagrams (aomdds) for graphical models. *J. Artif. Int. Res.*, 33(1):465–519, December 2008.

[38] C. C. Ribeiro and M. G. C. Resende. Path-relinking intensification methods for stochastic local search algorithms. *Journal of Heuristics*, 18(2):193–214, 2012.

[39] J. Sánchez-Oro, A. Martínez-Gavara, M. Laguna, A. Duarte, and A. Martí. Variable neighborhood scatter search for the incremental graph drawing problem. *Computational Optimization and Applications*, 68:775–797, 2017.

[40] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Trans. Syst. Man, Cybern.*, 11:109–125, 1981.

[41] J. Vanhatalo, H. Völzer, F. Leymann, and S. Moser. Automatic workflow graph refactoring and completion. In A. Bouguettaya, I. Krueger, and T. Margaria, editors, *Service-Oriented Computing – ICSOC 2008*, pages 100–115, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.