SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2000; **30**:1235–1284

Experimental studies on graph drawing algorithms

Luca Vismara[1*,†], Giuseppe Di Battista[2], Ashim Garg[3], Giuseppe Liotta[4], Roberto Tamassia[1] and Francesco Vargiu[5]

[1]*Center for Geometric Computing, Department of Computer Science, Brown University, 115 Waterman Street, Providence, RI 02912-1910, U.S.A.*
[2]*Dipartimento di Informatica e Automazione, Università degli Studi di Roma Tre, Via della Vasca Navale 79, 00146 Roma, Italy*
[3]*Department of Computer Science and Engineering, State University of New York at Buffalo, 226 Bell Hall, Buffalo, NY 14260-2000, U.S.A.*
[4]*Dipartimento di Ingegneria Elettronica e dell'Informazione, Università degli Studi di Perugia, Via Duranti 93, 06125 Perugia, Italy*
[5]*Centro Tecnico per la Rete Unitaria, Via Isonzo 21, 00198 Roma, Italy*

SUMMARY

Graph drawing plays an important role in the solution of many information visualization problems. Most of the graph drawing algorithms are accompanied by a theoretical analysis of their characteristics, but only extensive experimentations can assess the practical performance of graph drawing algorithms in real-life applications. In this paper, we describe the results of some of the most popular experimental studies on graph drawing algorithms. Each study presents an in-depth comparative analysis on a specific class of algorithms, namely, algorithms for orthogonal drawings, interactive algorithms, algorithms for hierarchical drawings, and force-directed and randomized algorithms. Copyright © 2000 John Wiley & Sons, Ltd.

KEY WORDS: graph drawing; diagram layout; experiments

1. INTRODUCTION

Many information visualization problems involve the drawing of a graph on the plane. Examples include the display of database conceptual models, class hierarchies in software engineering, project

*Correspondence to: Luca Vismara, Department of Computer Science, Brown University, 115 Waterman Street, Providence, RI 02912-1910, U.S.A.
†E-mail: lv@cs.brown.edu

Contract/grant sponsor: Consiglio Nazionale delle Ricerche, Project 'Geometria Computazionale Robusta con Applicazioni alla Grafica ed al CAD'
Contract/grant sponsor: ESPRIT Long Term Research Project 20244 (ALCOM-IT)
Contract/grant sponsor: National Science Foundation; contract/grant number: CCR–9732327
Contract/grant sponsor: National Science Foundation; contract/grant number: CDA–9703080
Contract/grant sponsor: U.S. Army Research Office; contract/grant number: DAAH04–96–1–0013

Received 2 July 1998
Revised 2 June 1999
Accepted 13 August 1999

management diagrams, and circuit schematics in electronics. Motivated by this type of application, a large body of graph drawing algorithms has been developed in the last two decades. A bibliographic survey [1], an annual symposium [2–7], various special issues of journals [8–10], and a recently published book [11] have been devoted to this research area.

A *graph drawing* algorithm receives as input a combinatorial description of a graph and returns as output a drawing of the graph. Various *drawing conventions* have been proposed for the representation of graphs in the plane. Usually, each vertex is mapped to a distinct point and each edge (u, v) is mapped to a simple Jordan curve between the points representing u and v. In particular, the edges are represented as polygonal chains in a *polyline* drawing, as chains of alternating horizontal and vertical segments in an *orthogonal* drawing, and as segments in a *straight-line* drawing. A *grid* drawing is embedded in a rectilinear grid such that the vertices and the bends of the edges have integer coordinates. In an *upward drawing*, each edge is drawn as a curve monotonically increasing in the y-direction. A *layered drawing* is a particular case of an upward drawing, where the vertices and edge-bends are placed on a set of horizontal layers. A *k-level* graph is a graph $G = (V, E)$, where the vertex set V can be partitioned into k disjoint subsets $V_1, \ldots, V_k$, called *levels*, and edges only connect vertices in different levels. In a *proper k-level* graph, edges only connect vertices in consecutive levels V_i and V_{i+1}. Usually, k-level graphs are drawn layered.

Within a given drawing convention, a graph has infinitely many drawings; however, one of the fundamental properties of a drawing is its *readability*, that is, the capability of conveying the information associated with the graph in a clear way. The readability of a drawing is expressed by means of *aesthetic criteria*, which can be formulated as optimization goals for the drawing algorithms. These aesthetic criteria have been selected through an extensive analysis of human-drawn diagrams from various contexts (see, e.g., [12]) and have been validated through experimental studies on human understanding of diagrams (see, e.g., [13,14]). Some aesthetic criteria are general, others depend on the drawing convention adopted and on the particular class of graphs considered (trees, planar graphs, hierarchical graphs, etc). They have been frequently used as quality measures in the evaluation and comparison of graph drawing algorithms.

While few experimental studies on graph drawing algorithms have been performed in the past, there is now a fast-growing interest in experimental comparative studies of this type. Application developers that have to select an algorithm for a particular type of drawing can compare the requirements of the application with the results of the study and have guidelines to decide which the most suitable algorithm for their purposes is.

Many papers presenting graph drawing algorithms show sample outputs from prototype implementations; some also provide limited experimental results on small test suites. However, only extensive experimentations can assess the practical performance of graph drawing algorithms in real-life applications. The properties that any good experimental study on algorithms should possess have been described by Johnson [15].

The first broad-view experimental study on graph drawing algorithms has been presented by Himsolt [16]. The author compares eleven algorithms devised for various types of graphs, namely, general graphs, directed acyclic graphs, planar graphs, and graphs with a special structure (trees, Petri nets, etc). The analysis of the experimental results follows a mixed qualitative and quantitative approach, providing an interesting description of the 'look and feel' of the drawings.

In this paper, we describe the results of some of the most popular experimental studies on graph drawing algorithms. For space reasons, we omit many details on the algorithms (the reader is referred

Table I. Algorithms under evaluation.

Drawing Convention	Algorithm	Section	References
Grid Orthogonal	*Bend-Stretch*	2.1	[25]
Grid Orthogonal	*Column*	2.1	[26]
Grid Orthogonal	*Giotto*	2.1	[27]
Grid Orthogonal	*Pair*	2.1	[28,29]
Grid Orthogonal	*No-Change*	3.2	[18,19,30]
Grid Orthogonal	*Relative-Coordinates*	3.2	[18,19,30]
Layered Straight-Line	*Barycentric*	4.1	[31]
Layered Straight-Line	*Median*	4.1	[32]
Layered Straight-Line	*Stochastic*	4.1	[33]
Layered Straight-Line	*GreedyInsert*	4.1	[34]
Layered Straight-Line	*GreedySwitch*	4.1	[34]
Layered Straight-Line	*Split*	4.1	[34]
Layered Straight-Line	*Assignment*	4.1	[35]
Layered Straight-Line	*Branch & Cut*	4.1	[22]
Layered Straight-Line	*Branch & Bound (Branch & Cut)*	4.1	[22]
Layered Straight-Line	*Iterative (Barycentric)*	4.1	[22]
Layered Straight-Line	*Iterative (Branch & Cut)*	4.1	[31]
Layered Spline	*Dot*	4.2	[36]
Layered Polyline	*Layers*	4.2	[31]
Grid Polyline	*Visibility*	4.2	[37,38]
Grid Polyline	*Lattice*	4.2	[39]
Straight-Line	*MagneticSpring*	5.1	[24]
Straight-Line	*Spring-FR*	5.2	[40]
Straight-Line	*Spring-KK*	5.2	[41]
Straight-Line	*SimulatedAnnealing-DH*	5.2	[42]
Straight-Line	*SimulatedAnnealing-T*	5.2	[43]
Straight-Line	*Randomized-GEM*	5.2	[44]

to the cited literature) and mainly focus on the description of the experimental results. Each study presents an in-depth comparative analysis of a specific class of algorithms: algorithms for orthogonal drawings [17] in Section 2, interactive algorithms [18,19] in Section 3, algorithms for hierarchical drawings [20–22] in Section 4, and force-directed and randomized algorithms [23,24] in Section 5.

The algorithms analyzed in these experimental studies are listed in Table I. For each algorithm, the drawing conventions it adheres to, the section of the paper in which it is presented, and the bibliographic references are given. The quality measures used in the experimental studies are listed in Table II, with the section of the paper in which they are introduced.

Table II. Quality measures analyzed.

Quality measure	Section
Area	2.1.2.1
Crossings	2.1.2.1
TotalBends	2.1.2.1
TotalLength	2.1.2.1
MaxBends	2.1.2.1
MaxLength	2.1.2.1
UnifBends	2.1.2.1
UnifLength	2.1.2.1
ScreenRatio	2.1.2.1
Time	2.1.2.1
AvgLength	3.3.1.1
AspectRatio	3.3.1.1
NewRows	3.3.1.1
NewColumns	3.3.1.1
NewBends	3.3.1.1
ResFactor	4.2.2.1
Area (ResFactor)	4.2.2.1
TotalLength (ResFactor)	4.2.2.1
MaxLength (ResFactor)	4.2.2.1
OrientationErrors	5.1.1.1
AvgOrientationAngle	5.1.1.1
VarOrientationAngle	5.1.1.1
VarLength	5.1.1.1
MinVertexDistance	5.1.1.1
LengthRatio	5.2.2.1

2. ORTHOGONAL DRAWINGS

2.1. Comparing four algorithms for orthogonal drawings

Orthogonal drawings are widely used for visualizing diagrams, such as Entity-Relationship and Data-Flow diagrams in database conceptual design, and circuit schematics in VLSI design. In this section, we present an extensive experimental study performed by Di Battista *et al.* [17] comparing four algorithms for orthogonal drawings.

2.1.1. The drawing algorithms under evaluation

The authors evaluate and compare the performance of four algorithms, denoted by *Bend-Stretch*, *Column*, *Giotto*, and *Pair*. These algorithms take as input general graphs (with no restrictions whatsoever on connectivity, planarity, etc) and construct orthogonal grid drawings.

Algorithms *Bend-Stretch* [25] and *Giotto* [27] are based on a general approach where the drawing is incrementally determined in three phases: the *planarization* phase determines the topology of the drawing; the *orthogonalization* phase computes an orthogonal shape for the drawing; the *compaction* phase produces the final drawing. This approach allows homogeneous treatment of a wide range of diagrammatic representations, aesthetics and constraints (see, e.g., [12,45,46]) and has been successfully used in industrial tools. The main difference between the two algorithms is in the orthogonalization phase: *Giotto* uses a network-flow method that guarantees the minimum number of bends but has quadratic time complexity; *Bend-Stretch* adopts the 'bend-stretching' heuristic that only guarantees a constant number of bends on each edge but has linear time complexity.

Algorithm *Column* is an extension of the orthogonal drawing algorithm by Biedl and Kant [26] to graphs of arbitrary vertex degree.

Algorithm *Pair* is an extension of the orthogonal drawing algorithm by Papakostas and Tollis [28] to graphs of arbitrary vertex degree. (For an improved version of the results presented in [28], see [29].)

Let n be the number of vertices of the input graph, m be the number of edges of the input graph, and c be the number of crossings in the constructed drawing. The worst-case asymptotic time complexity is $O(n+m)$ for algorithm *Column*, $O((n+m)\log(n+m))$ for algorithm *Pair*, and $O((n+c)^2\log(n+c))$ for algorithms *Bend-Stretch* and *Giotto*.

For the experimental study, the authors have used the implementations of *Bend-Stretch*, *Column*, *Giotto*, and *Pair* available in *Diagram Server* [47,48]. These implementations use methods that are efficient in practice but are not asymptotically worst-case optimal for tasks such as sorting, searching, and shortest path computations. Regarding algorithm *Bend-Stretch*, although the time complexity of the core algorithmic components is linear, the preliminary quadratic time planarization step that determines the overall time complexity is needed because the core algorithmic components take as input a planar graph. Note that it is NP-hard to compute the minimum number of crossings [49]; thus, the four algorithms heuristically attempt to reduce the number of crossings.

2.1.2. Experimental setting

2.1.2.1. Quality measures analyzed. The authors consider the following nine quality measures of a drawing:

Area: area of the smallest rectangle with horizontal and vertical sides covering the drawing;

Crossings: total number of edge-crossings;

TotalBends: total number of edge-bends;

TotalLength: total edge length;

MaxBends: maximum number of bends per edge;

MaxLength: maximum edge length;

UnifBends: standard deviation of the number of bends per edge;

UnifLength: standard deviation of the edge length;

ScreenRatio: deviation from the optimal aspect ratio, computed as the difference between the width/height ratio of the best of the two possible orientations (portrait and landscape) of the drawing and the standard 4/3 ratio of a computer screen.

It is widely accepted (see, e.g., [12–14]) that small values of the above measures are related to the perceived aesthetic appeal and visual effectiveness of the drawing. In addition, the authors consider the following quality measure of an algorithm:

Time: the running time.

2.1.2.2. Test suite. The test suite consists of 11 582 graphs,[‡] with number of vertices ranging from 10 to 100, which have been derived from a core set of 112 graphs used in 'real-life' software engineering and database application areas. In particular, the authors first collected 112 'real life' graphs with number of vertices between 10 and 100, from now on called *core graphs*, from the following sources: (i) software companies and large government organization; (ii) reference books in software engineering and database design, and articles on software visualization; (iii) theses in software and database visualization.

Then, the authors generated the graphs of the test suite as variations of the core graphs, devising a method for generating graphs 'similar' to the core graphs. The adopted scheme included the definition of several primitive operations for updating graphs, which correspond to the typical operations performed by designers of Entity-Relationship and Data-Flow diagrams, and the attribution of a certain probability to each of them.

At least 50 test graphs for each vertex cardinality between 10 and 100 were generated. The average number of edges of the test graphs is slightly higher than that of the core graphs.

Sparsity and 'near-planarity' are typical properties of graphs used in software engineering and database application areas [50]. As expected, the test graphs turn out to be sparse (the average vertex degree is about 2.7) and to have few edge-crossings (the experiments show that the average number of edge-crossings is no more than about 0.7 times the number of vertices). Graphs with more than 100 vertices were not included because they are rarely displayed in full in the above applications (clustering methods are typically used to hierarchically display large graphs).

2.1.2.3. Computing equipment. The experiments were performed on a Sun Sparc-10 workstation.

2.1.3. Analysis of the experimental results

Algorithms *Bend-Stretch*, *Column*, *Giotto*, and *Pair* have been executed on each of the 11 582 test graphs. Figures 1–3 show the average values of the nine quality measures. The x-axis of each chart indicates the number of vertices. The average values of the quality measures are computed over each group of graphs with number of vertices 10, 11, etc. The comparative analysis of the performance of the four algorithms for each quality measure is summarized below:

[‡]The graphs are available at `http://www.dia.uniroma3.it/people/gdb/wp12/undirected-1.tar.gz`.

Area (see Figure 1): *Giotto* outperforms *Bend-Stretch*, *Column*, and *Pair*. Two observations can be made: (i) While *Giotto* minimizes the number of bends, *Bend-Stretch*, *Pair* and *Column* do not. Since each bend occupies a unit cell on the grid, the drawings produced by *Bend-Stretch*, *Pair* and *Column* have in general larger area than the ones produced by *Giotto*. (ii) *Bend-Stretch*, *Pair* and *Column* transform the input graph into a biconnected graph. The (dummy) edges that are introduced by this phase are deleted only in the final drawing. In most cases, deleting dummy edges in a drawing does not reduce its overall area. Since biconnectivity is needed only for computing the st-numbering, the authors conjecture that the area of the drawings produced by *Bend-Stretch*, *Pair* and *Column* can be reduced by removing the dummy edges before the actual drawing of the graph is computed. They believe that this approach can also improve quality measures Crossings and TotalBends.

Crossings (see Figure 1): *Bend-Stretch* and *Giotto* behave more or less the same. *Bend-Stretch* is in general slightly worse than *Giotto*. The very different behavior of *Pair* and *Column* can be explained by considering that they do not perform a planarization phase; although in the worst case the number of crossings can be quadratic in the number of vertices, the planarization phase performed by *Bend-Stretch* and *Giotto* has a very good behavior in most cases.

ScreenRatio (see Figure 1): the behavior of *Giotto* is very good in the whole interval. The behavior of *Bend-Stretch* is about the same as that of *Giotto* for values of n between 40 and 100. The behavior of *Column* is unsatisfactory. The screen ratio of the drawings produced by the four algorithms changes very slowly with n for $n > 50$, and might indicate a convergence to some stable value. It is also interesting to note that the plots for *Column* and *Pair* converge towards each other as n increases. This is because both algorithms use similar techniques based on st-numbering and on the optimization of the use of columns. This is also reflected in the similar 'looks' of the drawings produced by these two algorithms.

TotalBends (see Figure 2): the experimental results fit accurately the theoretical results. Namely, *Giotto* has the minimum number of bends; *Bend-Stretch*, *Column*, and *Pair* have a number of bends that, allowing for the constants, is essentially the one predicted by the theoretical analysis.

MaxBends (see Figure 2): *Column* and *Pair* have almost identical behaviors and are the best among the four for $n > 35$. Indeed, their theoretical analysis shows that each edge has at most two bends. Regarding *Giotto* and *Bend-Stretch*, since the number of bends introduced by *Giotto* on each edge is theoretically unbounded, and since *Bend-Stretch* guarantees for planar graphs at most two bends on each edge, one would expect a better behavior of *Bend-Stretch* with respect to *Giotto*. The poor experimental performance of *Bend-Stretch* can be explained by noting that the edges involved in crossings are decomposed by the planarization phase into several fragments, and the bound of two bends per edge applies separately to each fragment. Finally, the average value of MaxBends is always less than four for *Giotto*, which suggests that the theoretical $O(n)$ worst-case value of MaxBends is rarely attained.

UnifBends (see Figure 2): *Giotto* has the best behavior here and outperforms *Bend-Stretch*, *Column*, and *Pair*. *Column* and *Pair* have almost identical behavior. It is interesting to observe that *Bend-Stretch* is better than *Column* and *Pair* for $n < 35$, while *Column* and *Pair* are better than *Bend-Stretch* for $n > 35$. This is consistent with the behavior for TotalBends, reflecting the fact

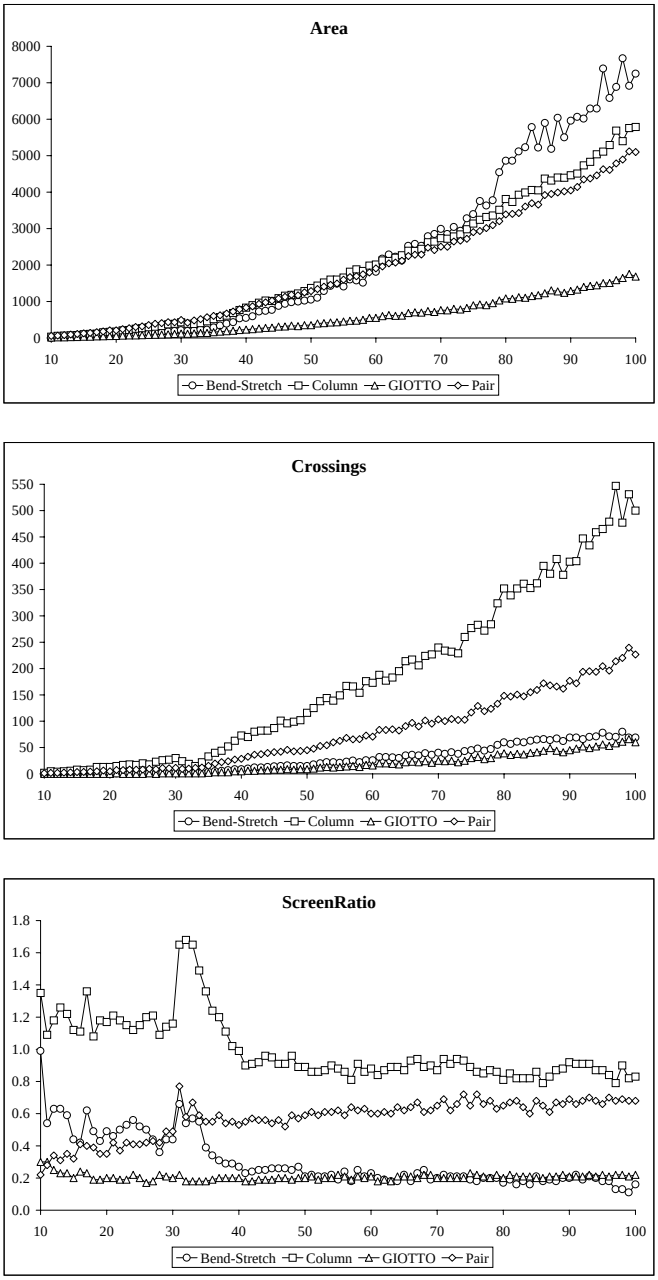

Figure 1. Comparison charts for Area, Crossings, and ScreenRatio: the x-axes indicate the number of vertices.

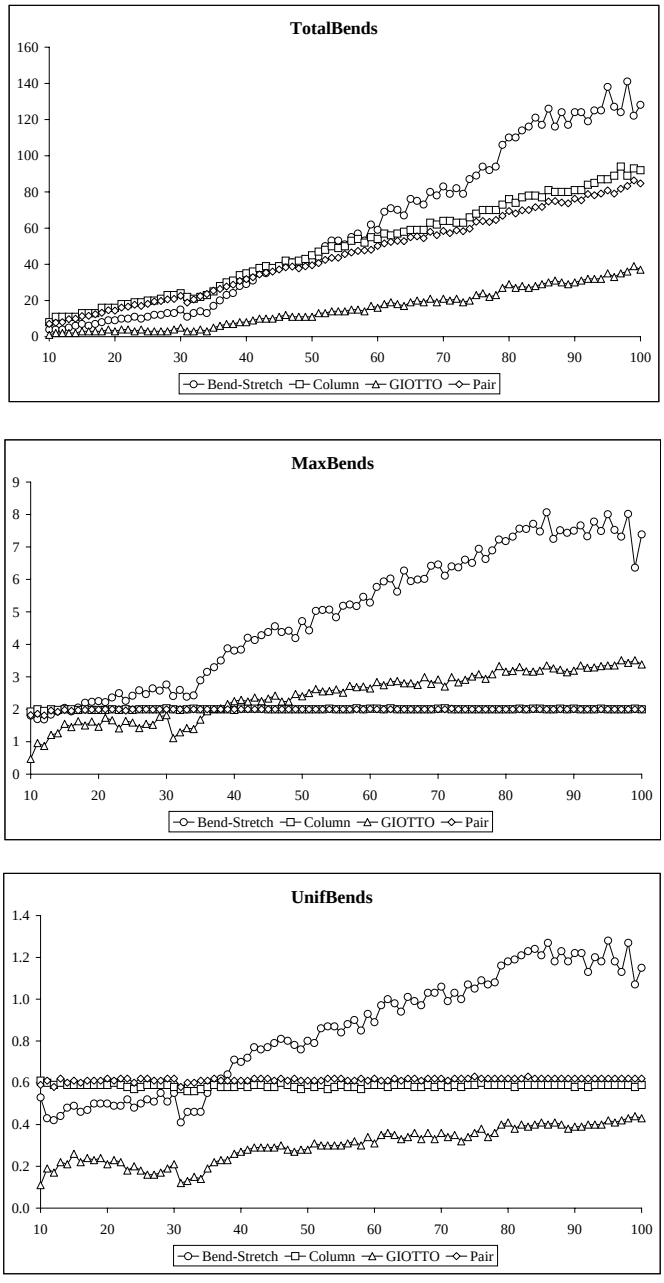

Figure 2. Comparison charts for TotalBends, MaxBends, and UnifBends: the x-axes indicate the number of vertices.

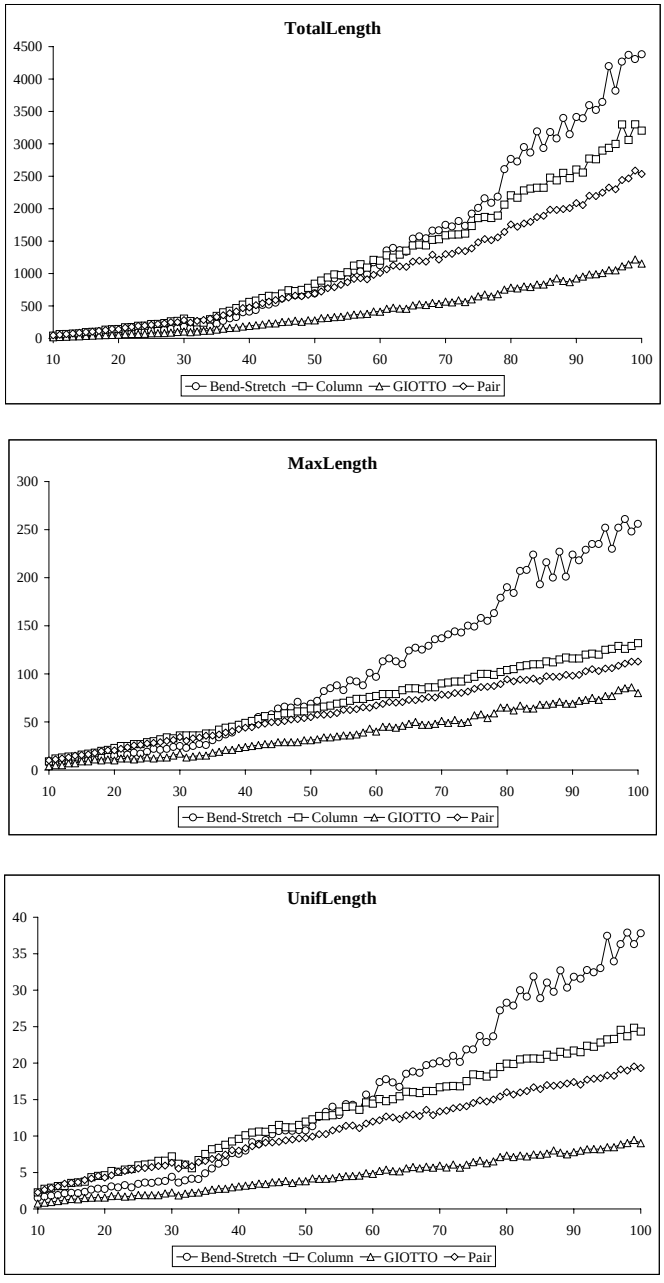

Figure 3. Comparison charts for TotalLength, MaxLength, and UnifLength: the x-axes indicate the number of vertices.

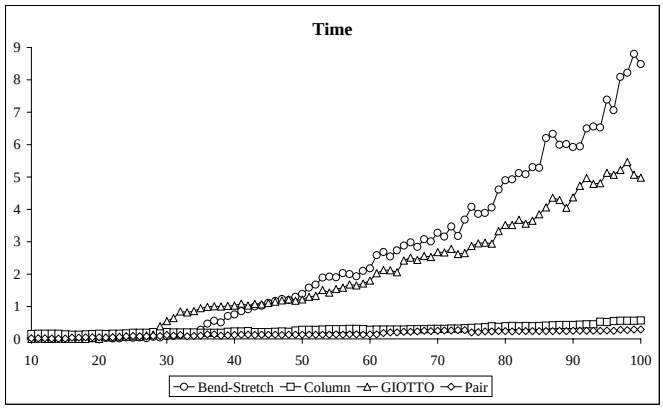

Figure 4. Comparison chart for Time (seconds): the x-axis indicates the number of vertices.

that UnifBends is numerically affected by the value of TotalBends. Also, note that, according to the theoretical analysis, *Column* and *Pair* have perfectly constant behaviors.

TotalLength (see Figure 3): *Giotto* is better than the other three algorithms. *Bend-Stretch* is better than *Column* for $n < 60$, while for $n > 60$ *Column* behaves better than *Bend-Stretch*. *Pair* always performs either the same or better than *Column*. Also note the similarity of the plots for TotalLength and Area, which fits with the intuitive notion that low TotalLength and Area generally go together. All the observations made for Area can be applied to this case.

MaxLength (see Figure 3): *Giotto* again has the best behavior. *Pair* and *Column* behave about the same. *Bend-Stretch* behaves better than *Pair* and *Column* for $n < 45$, whereas *Pair* and *Column* behave better than *Bend-Stretch* for $n > 45$. The authors believe that the length of the longest edge can be reduced for *Bend-Stretch*, *Pair*, and *Column* by implementing the st-numbering procedure based on a breadth-first search.

UnifLength (see Figure 3): *Giotto* performs much better than the other three algorithms. *Bend-Stretch* behaves better than *Pair* and *Column* for $n < 60$, whereas *Pair* and *Column* behave better than *Bend-Stretch* for $n > 60$. Note that the plots for UnifLength and for TotalLength are very similar. This reflects the fact that numerically UnifLength is affected by the value of TotalLength.

Time (see Figure 4): the implementations of the algorithms use methods that are efficient in practice but are not asymptotically optimal. *Column* and *Pair* are both quite fast, which is consistent with their low theoretical time complexities. Both *Bend-Stretch* and *Giotto* are much slower than *Column* and *Pair* for $n > 35$. Observe that *Bend-Stretch* is faster than *Giotto* for $n < 45$, whereas *Giotto* is faster than *Bend-Stretch* for $n > 45$. In the authors' opinion, this is the case

because for small graphs ($n < 45$) the quadratic time complexity of *Giotto* dominates the linear time complexity of *Bend-Stretch*, whereas for larger graphs ($n > 45$) *Bend-Stretch* is slower than *Giotto* for the following two reasons: (i) *Bend-Stretch* makes the graph biconnected by introducing dummy edges. Hence, the planarization phase operates on larger graphs for *Bend-Stretch* than it does for *Giotto*. (ii) Since the total number of bends introduced by *Bend-Stretch* is larger than the one introduced by *Giotto*, the tidy compaction phase [51,12] used by both algorithms, which treats bends as dummy vertices, requires more time for *Bend-Stretch* than it does for *Giotto*.

The plots relative to Time show a clear trade-off between running time and aesthetic quality of the drawings. Indeed, *Giotto* outperforms the other algorithms for most quality measures but it is considerably slower than *Column* and *Pair*.

The experiments provide a detailed quantitative evaluation of the performance of the four algorithms, and show that they exhibit trade-offs between 'aesthetic' properties (e.g., number of edge-crossings, number of edge-bends, edge length) and running time.

The main conclusion of the experimental study is that for a representative test suite of graphs derived from software engineering and database applications, algorithm *Giotto*, which is based on a preliminary planarization step followed by an exact bend minimization step, outperforms for most quality measures the other algorithms, which either do not perform a preliminary planarization step (*Column* and *Pair*) or use a heuristic bend minimization method (*Bend-Stretch*). However, it should be taken into account that *Giotto* is a much older drawing strategy whose steps have been extensively investigated in the last decade. Also, the benefits of using *Giotto* are paid for in terms of a substantially higher running time.

Also, the authors believe that some results for *Bend-Stretch*, *Pair*, and *Column* can be improved since the three algorithms all need to transform the input graph into a directed acyclic graph, and they do so by means of an st-numbering. The implementation of the st-numbering used in the experiments is based on a depth-first search, while in [29] it is shown that a procedure based on a breadth-first search can have a positive effect with respect to several quality measures, such as Area, Crossings, ScreenRatio, TotalLength, and MaxLength.

3. INTERACTIVE ALGORITHMS

Traditional graph drawing algorithms can also be called *one-shot* algorithms because, given a graph as input, they typically construct a drawing of the graph and then terminate. However, in several applications, such as software engineering and database design, users interact extensively with a displayed graph, continuously adding or deleting vertices and edges. Under such a scenario, a graph drawing system should update the drawing each time the displayed graph is modified by the user. Unfortunately, traditional drawing algorithms may not be suitable in these situations. Since they typically construct a drawing from scratch, they may fail to update the drawing quickly after the user modifies the displayed graph. Also, the new drawing constructed after the modification may be significantly different from the previous one, even if only a small change has been made in the displayed graph. In this case, the user's *mental map* [52,53], that is, the mental image the user has of the graph, is not preserved, and a considerable cognitive effort is required to correlate the new drawing and the

previous one. Hence, the need of devising techniques especially suited for such interactive settings. We survey some of these techniques in Section 3.1.

Note that the term *interactive* (or *dynamic*) graph drawing has a broad meaning, covering several aspects of graph visualization. In this section, we focus only on the problem of constructing geometric representations of a graph in a setting that allows addition or deletion of its vertices and edges. Also, throughout this section, *current graph* denotes the graph a user is currently interacting with, and *current drawing* denotes the drawing (of the current graph) currently displayed; by *update* we mean an atomic operation on the current graph, consisting of the addition and/or deletion of vertices and/or edges; *new graph* denotes the graph obtained after an update and *new drawing* denotes the drawing of the new graph.

3.1. Techniques for interactive graph drawing

Cohen *et al.* [54] have given efficient algorithms for drawing trees, series-parallel directed graphs, and planar st-digraphs in a dynamic environment. They maintain a data-structure that represents a drawing. Any update in the drawing is actually done on this data structure. Updating the data-structure generally takes $O(\log n)$ time, and hence is efficient. Their algorithm, however, does not attempt to preserve the mental map of the graph during updates.

Moen [55] has given an algorithm for dynamically constructing drawings of trees that uses subtree-contours (similar to those employed by Reingold and Tilford [56]) to efficiently update drawings.

Eades *et al.* [52] have given a technique for constructing hierarchical drawings of directed graphs, which aims at preserving the mental map of the graph. Newbery Paulish and Tichy [57], and Bohringer and Newbery Paulish [58] have used the algorithm of Sugiyama *et al.* [31] for constructing hierarchical drawings that maintain the relative ordering of vertices assigned to the same horizontal layer in successive drawings.

North [59] has defined two desirable properties of dynamic hierarchical graph drawings: *consistency* and *stability*. A consistent drawing is one that adheres to a particular layout style. Hence, a consistent hierarchical drawing of a tree always draws a parent above its children during any sequence of updates. A drawing is stable if making a small change in the displayed graph does not cause a large change in the drawing. Hence, stability is important if we wish to preserve the mental map of the graph. North [59] has developed a system called DynaDAG that incrementally constructs hierarchical drawings with these two properties.

Miriyala *et al.* [60] have given an algorithm for constructing an orthogonal drawing in a scenario where the user is allowed to make only edge insertions. Their algorithm keeps an already constructed drawing intact, and routes a newly added edge along a path with minimum number of edge-crossings.

Biedl *et al.* [61] have given a linear time algorithm for incrementally constructing orthogonal drawings with the restriction that a vertex cannot be moved after its initial placement.

Bridgeman *et al.* [62] have presented *InteractiveGiotto*, an interactive algorithm based on *Giotto* [27] for incrementally constructing orthogonal drawings. *InteractiveGiotto* allows the user to specify desired bends along the new edges, and thus gives him/her some control over the shape of the drawing. In a dynamic environment, *InteractiveGiotto* tries to maintain the mental map of the graph by preserving the following properties during an update: (i) the embedding of the graph, i.e., the circular ordering of the edges around each vertex; (ii) the edge-crossings; (iii) the edge-bends of the current drawing; (iv) for each vertex v, drawn as a rectangle R, the assignment of the incident edges of v to the

sides of R. In addition, *InteractiveGiotto* also has morphing capabilities that allow the transition from the current drawing to the new drawing to take place gradually on the screen.

Brandes and Wagner [63] have defined a Bayesian framework for dynamic layouts of graphs. They use a random field model for describing layouts. Under this model, each layout is assigned a probability that reflects the degree to which the layout conforms to a desired layout criteria. In addition, for each pair of layouts, L_1 and L_2, the authors define a probability that reflects the extent to which the mental map of the layout is preserved in the change from L_1 to L_2. Given a sequence of graphs (produced as a result of the user's updates), this formulation allows the authors to use Bayes' rule for conditional probabilities to find a sequence of layouts that compromises between preserving the user's mental map at each update and producing a final drawing that conforms to the desired layout criteria. This is a fairly broad framework that can be used to incorporate the need for preserving the user's mental map in several known traditional graph drawing algorithms, including Eades' spring embedder [64] and Tamassia's bend-minimization algorithm [27].

Papakostas and Tollis [30] have defined four scenarios for interactive orthogonal graph drawing, namely, *Draw-From-Scratch*, *Full-Control*, *No-Change*, and *Relative-Coordinates*, that formalize four ways by which a new drawing can be interactively constructed. Papakostas and Tollis [30], and Papakostas *et al.* [18] have theoretically analyzed the *No-Change* and the *Relative-Coordinates* scenarios, respectively. Papakostas *et al.* [18,19] have also conducted an extensive experimental study comparing the *No-Change* and *Relative-Coordinates* scenarios with respect to various readability and stability criteria. Note that [18] contains only a preliminary description of the experimental study, while the complete results and analysis of the experiments will appear in a future publication. In Section 3.2 we describe the four scenarios and the theoretical analyses, while in Section 3.3 we present the main results of the experimental study, as described in [18].

3.2. Interactive orthogonal graph drawing scenarios

We describe below the four scenarios defined in [30] that formalize the ways by which the new drawing can be constructed:

- *Draw-From-Scratch* Scenario: the new drawing is constructed from scratch by a traditional algorithm. The user has no control over the drawing and the new drawing can be significantly different from the current drawing.
- *Full-Control* Scenario: the user has complete control over the placement of the newly added vertices and/or edges.
- *No-Change* Scenario: vertices and edges do not change their positions after their initial placement.
- *Relative-Coordinates* Scenario: the general shape of the drawing remains the same after the update. The coordinates of some vertices and/or edges may change by a small constant because new rows and columns may be inserted in the middle of the drawing for placing newly added vertices and/or edges.

Papakostas and Tollis [30], and Papakostas *et al.* [18] theoretically analyze the *No-Change* and the *Relative-Coordinates* scenarios, respectively. They denote by $n(t)$ the number of vertices of the graph at time t, and by *update time* the time needed to construct the new drawing after the addition or deletion

of a single vertex or edge. They show that if we allow only additions of edges and vertices, and maintain a connected graph at every step, then:

- Under the *No-Change* scenario, it is possible to construct a drawing of a degree-4 graph with
  - Area at most $1.77[n(t)]^2$;
  - TotalBends at most $2.66n(t) + 2$;
  - at most 3 bends per edge;
  - $O(1)$ update time.
- Under the *Relative-Coordinates* scenario, it is possible to construct a drawing of a degree-4 graph with
  - Area at most $2.25[n(t)]^2$;
  - TotalBends at most $3n(t) - 1$;
  - at most 3 bends per edge;
  - a shift of at most 6 units horizontally and vertically for the coordinates of any vertex or bend after the insertion of a new vertex in the drawing.

3.3. Comparing no-change and relative-coordinates scenarios

Papakostas *et al.* [18,19] have conducted an extensive experimental study comparing the drawings constructed under the *No-Change* and *Relative-Coordinates* scenarios. They are interested in evaluating the readability and the stability of drawings constructed under these scenarios. The algorithms have been implemented on top of the *Graph Layout Toolkit* [65], a commercial system for graph layout and editing.

3.3.1. Experimental setting

3.3.1.1. Quality measures analyzed. The quality measures used in the experimental study can be divided into two groups, those used to ascertain the readability of the drawings and those used to ascertain the stability of the drawings. The first group consists of some of the quality measures described in Section 2.1.2.1, namely, Area, Crossings, TotalBends, and MaxLength, plus the following ones:

AspectRatio: width-to-height ratio of the drawing (note the slight difference between this quality measure and ScreenRatio described in Section 2.1.2.1);

AvgLength: average edge length.

The second group of quality measures consists of:

NewRows: number of rows added with the insertion of a vertex;

NewColumns: number of columns added with the insertion of a vertex;

NewBends: number of bends added with the insertion of a vertex.

3.3.1.2. Test suite. The test suite consists of 11 491 'real-life' graphs derived from those described in Section 2.1.2.2. In particular, a small number of edges and vertices were discarded in order to make the degree of all vertices at most four. Each graph has between 10 and 100 vertices, and is incrementally constructed by inserting one vertex at the time (together with its incident edges) into an initially empty graph. Each vertex (except the first one) is randomly selected among those adjacent to a vertex of the current graph, so that the graph is always connected.

3.3.2. Analysis of the experimental results

3.3.2.1. Readability. Figures 5 and 6 show the average values of the readability quality measures. The x-axis of each chart indicates the number of vertices. The average values of the quality measures are computed over each group of graphs with final number of vertices 10, 11, etc. The comparative analysis of the performance of the two scenarios is summarized below:

- The drawings produced are actually much better than what the worst-case theoretical analysis of [18,30] predicted. For example, for the *No-Change* scenario, Area and TotalBends are close to $0.5[n(t)]^2$ and $0.5n(t)$, respectively, compared to the theoretical worst-case bounds of $1.77[n(t)]^2$ and $2.66n(t)+2$, respectively. Similarly, for the *Relative-Coordinates* scenario, Area and TotalBends are close to $0.25[n(t)]^2$ and $0.25n(t)$, respectively, compared to the theoretical worst-case bounds of $2.25[n(t)]^2$ and $3n(t) - 1$, respectively.
- The *Relative-Coordinates* scenario produces drawings with lower Crossings, AvgLength, and MaxLength than the *No-Change* scenario. This is expected since in the *Relative-Coordinates* scenario a new vertex is placed close to its adjacent vertices already present in the drawing (at the cost of inserting new rows or columns within the drawing), whereas, in the *No-Change* scenario, modifications of the existing drawing are not allowed and hence the new vertex may be placed far from its adjacent vertices, thus causing more edge-crossings and increasing average and maximum edge lengths.
- AspectRatio is quite good for both scenarios, and the average drawing has a squarish shape.
- For both scenarios, Area and Crossings seem to have a quadratic growth, while TotalBends, MaxLength, and AvgLength seem to have a linear growth.

3.3.2.2. Stability. Table III shows the average values of the stability quality measures computed over the entire test suite. For both scenarios, the average numbers of new rows, new columns, and new bends added with the insertion of a degree-1, degree-2, degree-3, and degree-4 vertex are given. As noted in [18,19], small values of the stability quality measures for degree-2 vertex insertions have a positive impact on the readability quality measures Area and TotalBends. Hence, the good stability performance of the *Relative-Coordinates* scenario for degree-2 vertex insertions explains why this scenario obtains better results for Area and TotalBends than the *No-Change* scenario.

In conclusion, the experimental study on the *No-Change* and *Relative-Coordinates* scenarios indicates that their actual performances are much better than what the worst-case theoretical analysis predicted. Few columns, rows, and bends are added with each vertex insertion under both scenarios. The main result of the study is that the drawings produced under the *Relative-Coordinates* scenario are considerably better than those produced under the *No-Change* scenario for both readability and stability quality measures.

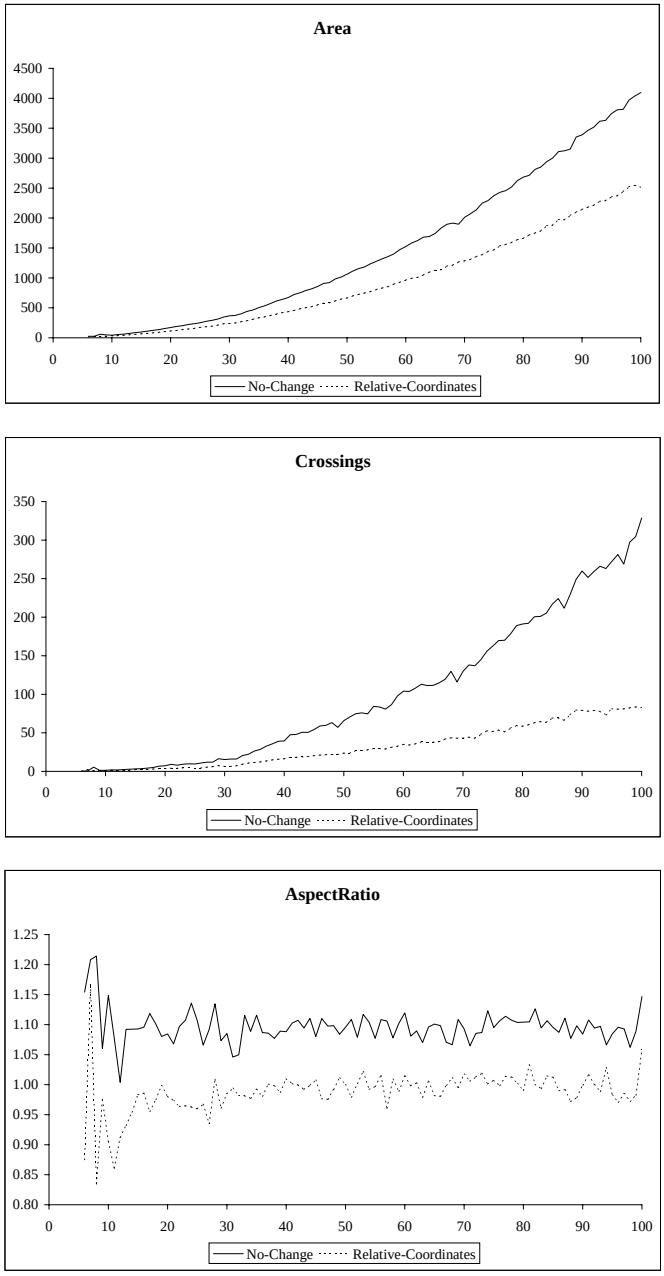

Figure 5. Comparison charts for Area, Crossings, and AspectRatio: the x-axes indicate the number of vertices. (Data courtesy of Achilleas Papakostas, Janet M. Six and Ioannis G. Tollis.)

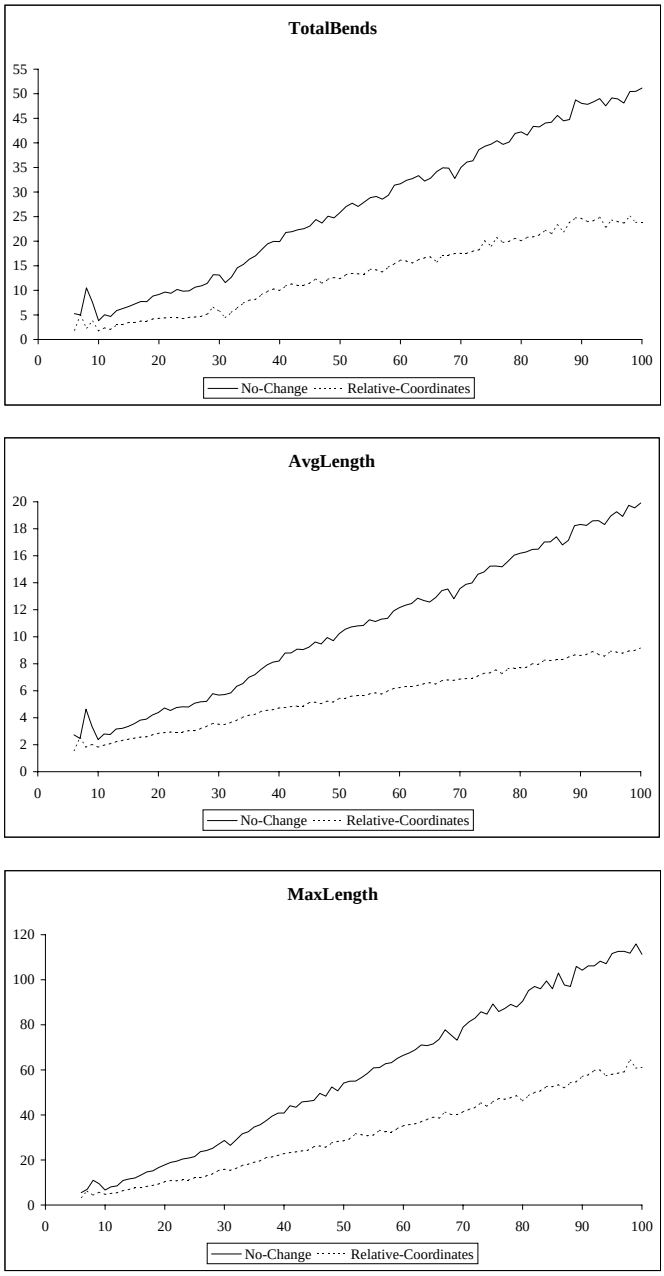

Figure 6. Comparison charts for TotalBends, AvgLength, and MaxLength: the x-axes indicate the number of vertices. (Data courtesy of Achilleas Papakostas, Janet M. Six and Ioannis G. Tollis.)

Table III. Average values of NewRows, NewColumns, and NewBends computed over the entire test suite. (Data courtesy of Achilleas Papakostas, Janet M. Six and Ioannis G. Tollis.)

Degree	NewRows	NewColumns	NewBends
	No-change		
1	0.551	0.605	0.118
2	0.926	0.953	2.050
3	1.102	1.135	3.205
4	1.660	1.730	5.022
	Relative-coordinates		
1	0.515	0.485	0
2	0.337	0.435	0.812
3	0.707	0.740	2.447
4	1.343	1.373	3.987

4. HIERARCHICAL DRAWINGS

Directed acyclic graphs (DAGs) are commonly used to model hierarchical structures such as PERT[§] diagrams in project planning, class hierarchies in software engineering, and is-a relationships in knowledge-representation systems. It is customary to represent these graphs so that all the edges flow in the same direction, e.g., from top to bottom or from left to right.

Algorithms for drawing directed graphs can be classified into two categories based on their approach to constructing drawings.

Layering-Based: these algorithms construct layered drawings; they accept, as input, directed graphs without any particular restriction (the input directed graph can be planar or not, acyclic or cyclic). Note that in layered drawings, even though vertices and edge-bends are placed at integer coordinates, the edge-crossings can be arbitrarily close to each other or to the vertices and edge-bends. Layering-based algorithms generally follow the methodology of Sugiyama *et al.* [31], which consists of the following steps:

1. Remove existing cycles in the input graph by reversing the direction of some edges.
2. Assign vertices to layers heuristically optimizing some criteria, such as the total edge length.
3. Introduce fictitious vertices along edges whose end-vertices are not on consecutive layers. The result is a proper k-level graph. (For a detailed analysis of this step, see [66].)

[§]PERT is the acronym of the Program Evaluation and Review Technique used for planning and controlling the execution times of the various tasks of a complex project.

4. Reduce the crossings among edges by permuting the order of vertices on each layer (see Section 4.1).
5. Remove the fictitious vertices introduced in Step 3, replacing them with edge-bends.
6. Reduce the number of bends by readjusting the position of vertices on each layer.

Because of their generality and conceptual simplicity, these algorithms are very popular among the designers of practical graph drawing systems. Several layering-based algorithms have been designed (see [1,11] for a detailed bibliography). The above steps have also been investigated separately, and various heuristics have been proposed for each of them (see again [1,11]).

Grid-Based: These algorithms accept, as input, a planar st-digraph, i.e., a planar DAG with exactly one source and one sink, and construct an upward grid drawing of it. Although the requirement of having just one source and one sink may appear too restrictive, these directed graphs occur in several practical applications, such as activity planning (where they are called PERT graphs), network flows, etc. Note that in upward grid drawings the vertices, the edge-bends, and the edge-crossings are all placed at integer coordinates.

These algorithms are also called *numbering-based* algorithms because they typically construct a numbering of the vertices and faces of the planar st-digraph, and compute the coordinates of the vertices and bends using this numbering.

The grid-based algorithms have two advantages: first, their performance on planar st-digraphs has been theoretically analyzed, and second, their running times are usually low. The disadvantage is that a nonplanar DAG needs to be converted into a planar st-digraph, before it can be drawn using these algorithms. This is done by introducing a fictitious vertex for each crossing between two edges. These fictitious vertices are assigned a position on the grid, but are not represented in the final drawing.

4.1. Crossing minimization in layered drawings

As mentioned above, the six steps of the layering-based algorithms have also been investigated separately, and various heuristics have been proposed for each of them (see [1,11]). Of particular importance, as for any type of drawing (see, e.g., [13]), is Step 4, the minimization of the number of edge-crossings. Various experimental results on heuristics for Step 4 have been presented (see [1,11]).

In this section we describe experimental results for the 2-*layer crossing minimization* problem, where the vertices belong to two consecutive layers of the drawing and the edges are represented as straight-line segments. This problem has been shown to be NP-complete [49], even if the permutation of the vertices on one layer is fixed [67,32]. In particular, we present the experimental results described by Jünger and Mutzel in [22], which confirm and extend those previously described, e.g., in [34,68].

4.1.1. The drawing algorithms under evaluation

The authors first study the restricted version of the 2-layer crossing minimization problem where the permutation of the vertices on one layer is fixed. In the rest of the section, we refer to this restricted problem as the *one-sided* crossing minimization problem. The authors present a branch-and-cut algorithm for the computation of the optimal solution. Using this algorithm, they are

able to experimentally assess the performance of the following popular heuristics: the barycentric heuristic [31], the median heuristic [32], the stochastic heuristic [33], the greedy-insert, greedy-switch and split heuristics [34], and the assignment heuristic [35].

The general 2-layer crossing minimization problem is then considered. In the rest of the section, we refer to the general problem as the *two-sided* crossing minimization problem. The authors present a branch-and-bound algorithm for the computation of the optimal solution, which uses the previous branch-and-cut algorithm for the one-sided crossing minimization problem as a subroutine. Alternatively, a heuristic approach consists of an iterative process: choose a permutation of the vertices on the first layer, apply one of the heuristics or the branch-and-cut algorithm for the one-sided crossing minimization problem to the second layer, then use the result to apply one of the heuristics or the branch-and-cut to the first layer, and so on, back and forth, until the crossing number is no longer reduced, that is, a local minimum is found. This approach is also used to compute an initial solution for the branch-and-bound algorithm.

4.1.2. Experimental setting

4.1.2.1. Quality measures analyzed. The quality measures considered in this experimental study are Crossings and Time, defined in Section 2.1.2.1. In particular, the value of Crossings obtained with the heuristic methods is compared with the optimal value obtained with the branch-and-cut or the branch-and-bound algorithms.

4.1.2.2. Test suite. For the one-sided crossing minimization problem, three experiments have been performed. The first one on 900 randomly generated graphs with 20 vertices on each layer and with increasing density. The second one on 100 randomly generated sparse graphs with the number of vertices on each layer increasing from 10 to 100 and average degree 2. The third one on 6 particular graphs having $3 \leq k \leq 8$ vertices on the fixed layer and $2^k - 1$ vertices on the other.

For the two-sided crossing minimization problem, four experiments have been performed. The first two on 900 randomly generated graphs with 10 vertices on each layer and with increasing density. The second two on 100 randomly generated sparse graphs with the number of vertices on each layer increasing from 10 to 100 and average degree 2.

4.1.2.3. Computing equipment. The experiments were performed on a Sun Sparc-10 workstation.

4.1.3. Analysis of the experimental results

The authors first consider the one-sided crossing minimization problem. A surprising result of the first experiment is that only the barycentric and the median heuristics are slightly faster than the branch-and-cut algorithm (see Figure 7). And between these two heuristics, the barycentric is consistently more accurate (see Figure 7). Thus, for problems of small to moderate size the running time of the branch-and-cut algorithm is definitely reasonable. In the second experiment, the barycentric and the median heuristics are significantly faster than the branch-and-cut algorithm (see Figure 8), and the barycentric heuristic is again the most accurate (see Figure 8). The third experiment basically confirms the results of the previous two: the barycentric heuristic is the fastest, while the stochastic heuristic is the most accurate; for $k \leq 7$, however, the running time of the branch-and-cut algorithm is acceptable.

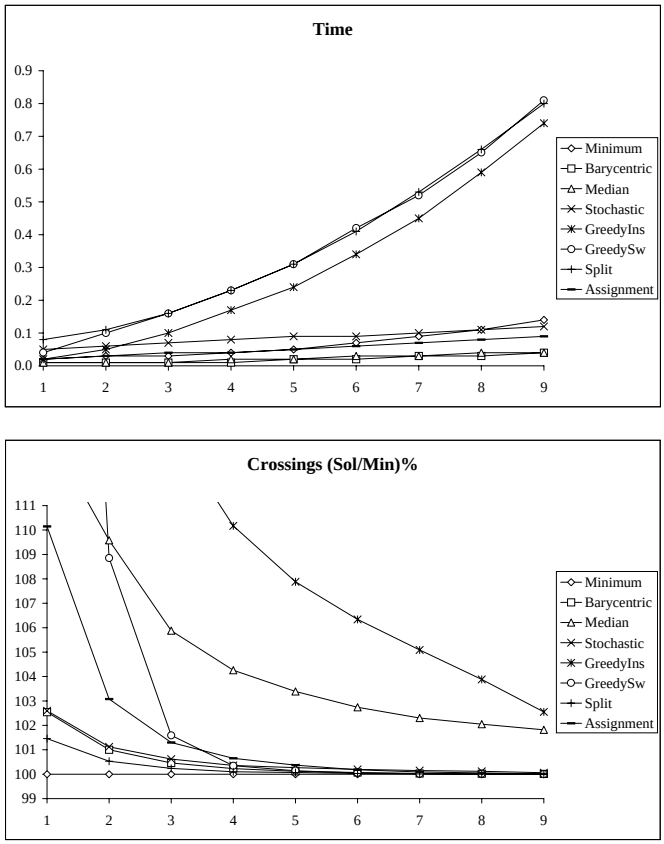

Figure 7. Comparison charts for Time (seconds) and Crossings in the first experiment relative to the one-sided crossing minimization problem: the x-axes indicate the density of the graph. (Data courtesy of Michael Jünger and Petra Mutzel.)

The authors then consider the two-sided crossing minimization problem. In the first experiment, as one may expect, the branch-and-bound algorithm running time increases rapidly with the density of the graph, while the iterative algorithms are not very sensitive to density (see Figure 9, where the plot of the branch-and-bound algorithm in the Time chart is completely out of scale). However, the optimal solution is usually much better than the solution of any iterative algorithm, especially for sparse graphs. Among the iterative algorithms the best results are obtained by the one based on the barycentric heuristic, which sometimes outperforms the one based on the branch-and-cut algorithm (see Figure 9). The second experiment shows that a sensible improvement of the solution found by any iterative algorithm can be obtained by repeating the algorithm a few times with different random initial permutations for the first layer and then choosing the best result (see the bottom chart of Figure 9). In

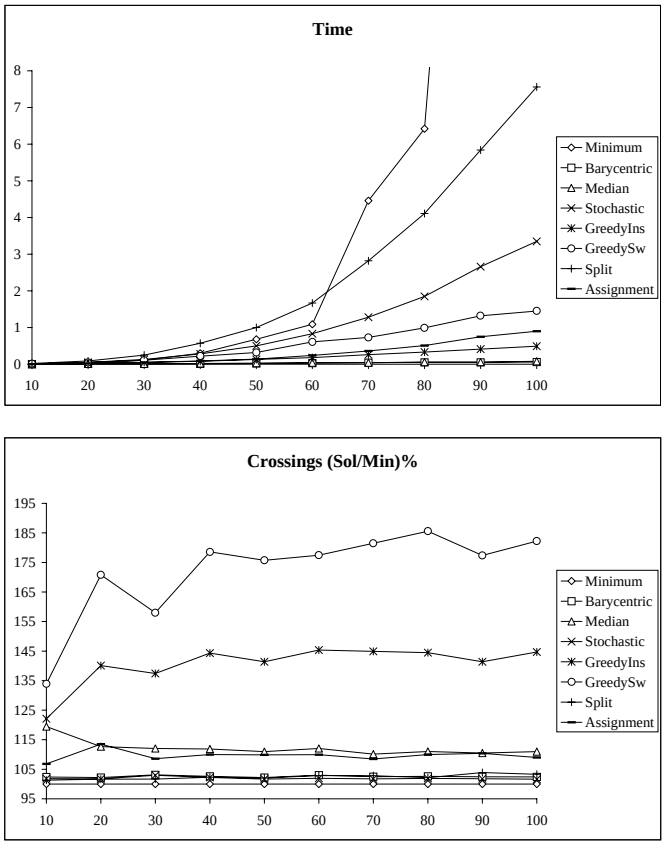

Figure 8. Comparison charts for Time (seconds) and Crossings in the second experiment relative to the one-sided crossing minimization problem: the x-axes indicate the number of vertices. (Data courtesy of Michael Jünger and Petra Mutzel.)

the third experiment, the size of the problems makes the branch-and-bound algorithm impractical. The best iterative algorithm, in terms of quality of the solution and running time, is the one based on the barycentric heuristic (see Figure 10). Contrary to the case of graphs with increasing density, the fourth experiment shows that for sparse graphs only a slight improvement of the solution can be obtained by repeating the algorithm a few times with different random initial permutations for the first layer and then choosing the best result.

The results of the computational experiments described above can be summarized as follows:

- When the permutation of the vertices on one layer is fixed, the exact minimum crossing number can be efficiently computed in practice, so there is no real need for heuristics.

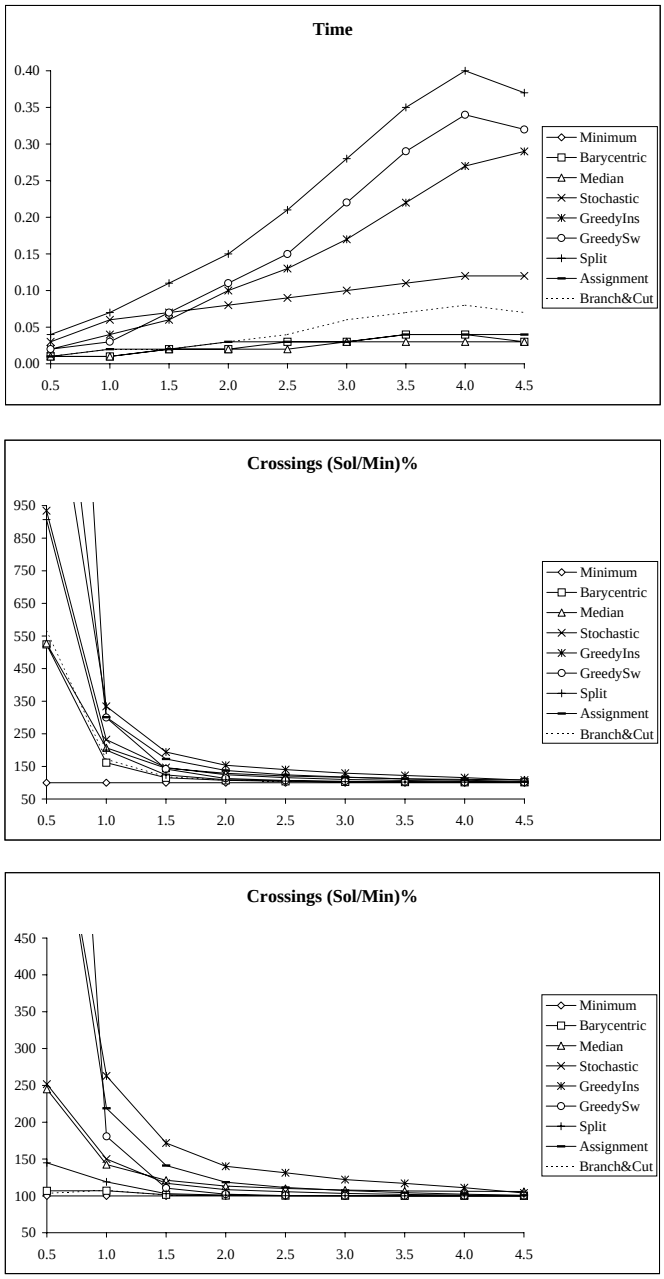

Figure 9. Comparison charts for Time (seconds) and Crossings in the first and second experiments relative to the two-sided crossing minimization problem: the x-axes indicate the density of the graph. In the Time chart, the plot of the branch-and-bound algorithm is completely out of scale. (Data courtesy of Michael Jünger and Petra Mutzel.)

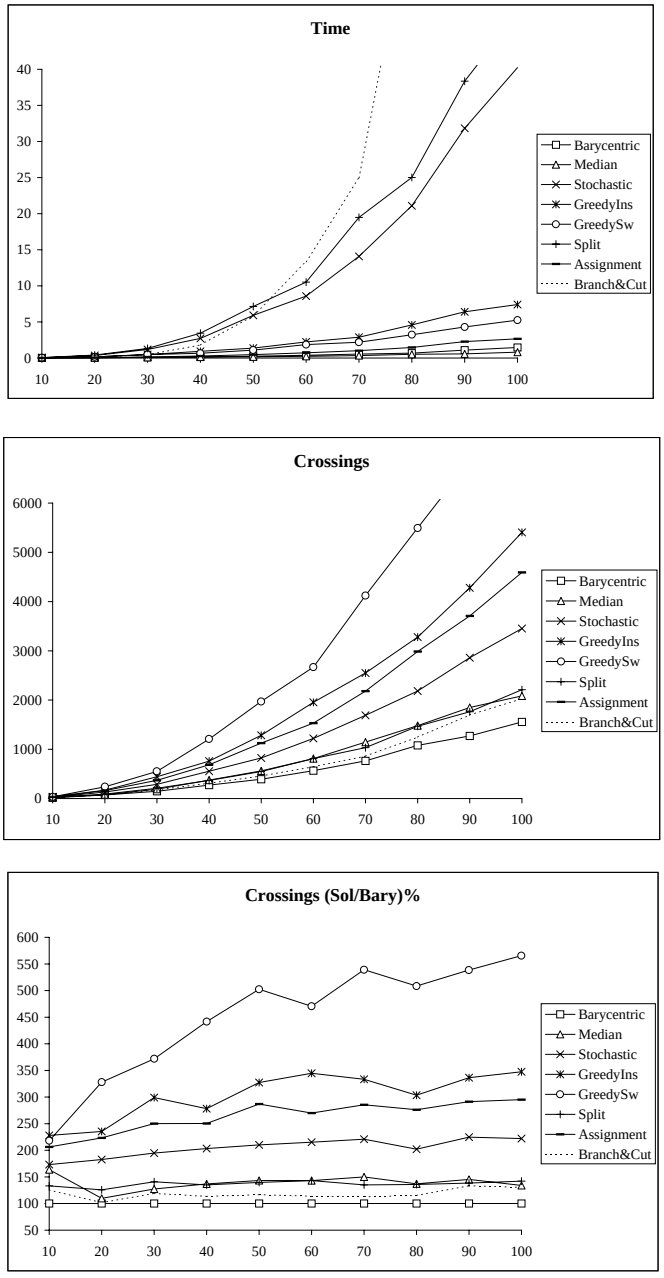

Figure 10. Comparison charts for Time (seconds) and Crossings in the third experiment relative to the two-sided crossing minimization problem: the x-axes indicate the number of vertices. (Data courtesy of Michael Jünger and Petra Mutzel.)

- In the general case, the exact minimum crossing number can be efficiently computed only for small sparse graphs in which the smaller sized layer contains at most 15 vertices. For dense and larger graphs, the iterative algorithm based on the barycentric heuristic is clearly the algorithm of choice, especially if repeated a few times with different random initial permutations for the first layer.

Two branch-and-cut algorithms for the related 2-level planarization problem and for the multi-layer crossing minimization problem have been presented, accompanied by some experimental results, by Mutzel [69,70] and by Jünger *et al.* [71], respectively.

4.2. Comparing four algorithms for hierarchical drawings

An extensive experimental study on hierarchical drawings is presented by Di Battista *et al.* in [20,21]. The authors consider the important class of directed acyclic graphs (DAGs), and compare the performance of four drawing algorithms specifically developed for them.

4.2.1. The drawing algorithms under evaluation

The authors evaluate and compare the performance of two layering-based algorithms, *Dot* and *Layers*, and two grid-based algorithms, *Visibility* and *Lattice*.

Dot is a highly optimized algorithm,[¶] developed by Koutsofios and North [36] as a successor to *Dag* [72,73]. *Dot* first constructs a polyline layered drawing of the input directed graph and then, as a final step, converts the polygonal chains representing the edges into smooth curves using splines. However, since all other algorithms considered in this study represent edges as polygonal chains, the authors have decided to analyze the polyline drawing produced by *Dot* and not the final drawing with curved lines.

Layers is the original algorithm by Sugiyama *et al.* [31]. For the study, the authors have used the implementation of *Layers* available in *GDW* [74], a system for prototyping and testing graph drawing algorithms.

The grid-based algorithms whose performance the authors evaluate and compare fall into two categories:

Visibility Representation-Based: These algorithms use a two-step process for constructing drawings [37,38]. In the first step, they construct a *visibility representation* of the input planar st-digraph [75,76]. (In a visibility representation, vertices and edges are represented as horizontal and vertical line-segments, respectively; two vertices are connected by an edge if and only if they are visible to each other.) In the second step, they construct a polyline drawing of the planar st-digraph from the visibility representation; this is done by replacing each vertex-segment with a point and by approximating each edge-segment with a polygonal line containing at most two bends. A *topological numbering* of a DAG is such that for every directed edge (u, v), v is assigned a higher number than u. The visibility representation is constructed using two

[¶]An implementation of *Dot* is available at `http://www.research.att.com/sw/tools/graphviz/`.

numberings [76,77]: a topological numbering of the vertices of the planar st-digraph, and a topological numbering (in the dual graph) of the faces of the planar st-digraph.

The authors evaluate the performance of one algorithm, called *Visibility*, which follows this approach, and of three variations of it, called *Barycentric Visibility*, *Long Edge Visibility*, and *Median Visibility*. For the study they have used the implementations of these algorithms available in *GDW*. The differences among Algorithms *Visibility*, *Barycentric Visibility*, *Long Edge Visibility*, and *Median Visibility* are in the strategy they use for substituting the vertex and edge-segments of the intermediate visibility representation with the points and polygonal chains of the final drawing; they put the vertex in the middle point of the vertex-segment, in the barycenter of the endpoints of the incident edge-segments, on the endpoint of the longest incident edge-segment, and on the median of the endpoints of the incident edge-segments, respectively.

Poset-Based: These algorithms view planar st-digraphs as covering graphs[‖] of partially ordered sets (*posets*). They exploit the relationship between the upward planarity of DAGs and the order-theoretic properties of planar lattices (see, e.g., [78,79]).

The authors evaluate the performance of one poset-based algorithm: the dominance drawing algorithm of [39], called *Lattice* in this study. This algorithm computes two topological numberings of the vertices of the input planar st-digraph; one numbering gives the x-coordinates of the vertices and bends, and the other gives the y-coordinates. These numberings are obtained by scanning the outgoing edges of each vertex of the planar st-digraph in the left-to-right and right-to-left order, respectively. For the study the authors have used the implementation of *Lattice* available in *GDW*.

For all the grid-based algorithms, the simple planarization method used in the study is the one described in [37].

4.2.2. Experimental setting

4.2.2.1. Quality measures analyzed. The quality measures considered for drawings of DAGs are those described in Section 2.1.2.1 (except UnifBends, UnifLength, and Time), plus the following one:

ResFactor: Inverse of the minimum distance between two vertices, or two edge-crossings, or an edge-crossing and a vertex.

The issue of the *resolution* of a drawing has been extensively studied, motivated by the finite resolution of physical rendering devices. Several papers have been published about the resolution and the area of drawings of graphs (see, e.g., [39, 80–82]). The resolution of a drawing is defined as the minimum distance between two vertices. The grid-based algorithms consider edge-bends and edge-crossings as 'dummy' vertices for computing the resolution. The layering-based algorithms, instead,

[‖] The covering graph of a partially ordered set $\{S, \prec\}$ is a directed graph G whose vertices correspond to the elements of S, and such that (u, v) is an edge of G if and only if $u \prec v$ and there is no other element x of S such that $u \prec x \prec v$.

do not consider the edge-crossings as dummy vertices for computing the resolution. Since the quality measures Area, TotalLength, and MaxLength of a drawing depend on its resolution, two drawings can be compared for these measures only if they have the same resolution. ResFactor allows one to scale a drawing Γ_1 produced by a layering-based algorithm so that it has the same resolution as that of a drawing Γ_2 produced by a grid-based algorithm; the scaling factor is equal to R_1/R_2, where R_1 and R_2 are the value of ResFactor for Γ_1 and Γ_2, respectively.

4.2.2.2. Test suite. The experimental study is performed on two different sets of DAGs,** both with a strong connection to 'real-life' applications. The authors consider two typical contexts where DAGs play a fundamental role, namely, software engineering and project planning.

The test DAGs of the first set are referred to as *North DAGs*. They were obtained from a collection of directed graphs [83] that North collected at AT&T Bell Labs by running for two years *Draw DAG*, an e-mail graph drawing service that accepts directed graphs formatted as e-mail messages and returns messages with the corresponding drawings [36].

Originally, the North DAGs consisted of 5114 directed graphs, whose number of vertices varied in the range 1 . . . 7602. However, the density of the directed graphs with a number of vertices outside the range 10 . . . 100 was very low; since these directed graphs represented a very sparse statistical population, the authors decided to discard them. Then they noted that many directed graphs were isomorphic; since the vertices of the directed graphs have labels associated with them, the problem of recognizing isomorphic graphs is tractable. For each isomorphism class, they kept only one representative directed graph. Also, they deleted the directed graphs where subgraphs were specified as clusters, to be drawn in their own distinct rectangular region of the layout, because constrained algorithms were beyond the scope of the study. After this filtering, 1277 directed graphs were left.

Still, 491 directed graphs were not connected and this was a problem for running the algorithms implemented in *GDW* (they assume input directed graphs to be connected). Instead of discarding these directed graphs, the authors followed a more practical approach, by randomly adding to each directed graph a minimum set of directed edges to make it connected. Finally, they made the directed graphs acyclic, where necessary, by applying some heuristics for inverting the direction of a small subset of edges.

The authors then performed a first set of experiments and produced the statistics by grouping the DAGs by number of vertices. Although the comparison among the algorithms looked consistent (the produced plots never oddly overlapped), each single plot was not satisfactory, because it showed peaks and valleys. They went back to study the test suite and observed that grouping them by number of vertices was not the best approach. In fact, the North DAGs come from very heterogeneous sources, mainly representing different phases of various software engineering projects; as a result, directed graphs with more or less the same number of vertices may be either very dense or very sparse.

Since most of the analyzed quality measures strongly depend on the number of edges of the DAG (e.g., Area, TotalBends, and Crossings), the authors decided that a better approach was to group the DAGs by number of edges. After some tests, they clustered the DAGs into nine groups, each with at least 40 DAGs, so that the number of edges of the DAGs belonging to group i, $1 \leq i \leq 9$, is in the

**The two sets of DAGs are available at `http://www.dia.uniroma3.it/people/gdb/wp12/directed-acyclic-1.tar.gz`.

range $10i \ldots 10i + 9$. The resulting test suite consists of 1158 DAGs, each with edges in the range $10 \ldots 99$.

The test DAGs of the second set are referred to as *Pert DAGs*. Although such DAGs have been randomly generated by one of the facilities of *GDW*, their construction is based on refinement operations typical of project planning.

First, the authors generated a set of skeleton planar DAGs containing a small number of vertices to simulate the initial models of the projects. This was done by randomizing an ear-composition[††] for each DAG. Second, they performed a random sequence of typical project planning refinement steps, i.e., expanding an edge into a series and/or a parallel component and inserting new edges between existing vertices. The inserted edges represent precedences between activities that were not captured by the initial models.

The resulting test suite consists of 813 DAGs with edges in the range $10 \ldots 150$ and vertices in the range $10 \ldots 100$. As for the North DAGs, the Pert DAGs were grouped by number of edges, so that the number of edges of the DAGs belonging to group i, $1 \leq i \leq 14$, is in the range $10i \ldots 10i + 9$.

The Pert DAGs are generally denser than the North DAGs and they are single-source single-sink. As shown in the next section, there are some quality measures for which the relative performance of the algorithms is different for the North DAGs and for the Pert DAGs. Also, the plots obtained for the Pert DAGs are in general smoother, reflecting the relative uniformity of the statistical population.

4.2.3. Analysis of the experimental results

Algorithms *Dot*, *Layers*, *Visibility* (with its variations *Barycentric Visibility*, *Long Edge Visibility*, *Median Visibility*), and *Lattice* have been executed on every North DAG and Pert DAG, and the data for all eight quality measures have been collected. Because of the different nature of the two test suites, the authors compare the performance of the algorithms for the North DAGs and the Pert DAGs separately. In addition, since the quality measures Area, TotalLength, and MaxLength depend upon the resolution of the drawings, they compare the layering-based and the grid-based algorithms separately for these three quality measures. The other five quality measures do not depend upon the resolution, so for them all four algorithms are compared together. Figures 11–16 show the average values of the quality measures; the left columns of these figures contain the charts for the North DAGs, and the right columns contain those for the Pert DAGs. The x-axis of each chart indicates the number of edges. The average values of the quality measures are computed over each group of DAGs with number of edges in the range $10 \ldots 19$, $20 \ldots 29$, etc.

The authors start the experimental analysis by comparing the behavior of *Visibility* and of its variations *Barycentric Visibility*, *Long Edge Visibility* and *Median Visibility*. As a result, they find that the behavior of the visibility representation-based algorithms is almost identical for all the quality measures, with *Visibility* performing slightly better than the others. In order to improve the readability of the other charts and to simplify the presentation of the experimental results, *Visibility* is used as the representative visibility representation-based algorithm.

[††]Informally speaking, an ear-composition of a graph G is an incremental construction of G (starting from an initial subgraph) obtained by adding a simple path at the time, such that, at each step, the added path has only the end-vertices in common with the current graph.

The comparative analysis of the performance of the four algorithms for each quality measure and for each set of input DAGs is summarized below:

Area (see Figure 11): *Dot* performs better than *Layers*, and *Lattice* performs better than *Visibility* for both the North DAGs and the Pert DAGs. While for the North DAGs, the plots grow linearly for edge numbers in the range 10 . . . 60, for the Pert DAGs they show quadratic growth in the entire range. Also, the difference in the performance of the two grid-based algorithms is significant for DAGs with more than 75 edges, whereas the two layering-based algorithms perform about the same in the entire range.

ResFactor (see Figure 12): not surprisingly, ResFactor is equal to 1 for the grid-based algorithms in the entire range. On the other hand, the layering-based algorithms tend to have a non-constant ResFactor. This reflects the fact that they do not take edge-crossings into consideration for defining the resolution. The bottom charts of Figure 12 show a comparative study of the area of the drawings produced by the four algorithms. Note that there are two plots for each layering-based algorithm. They show two different measures for the area: one takes ResFactor into account, and the other does not. Note that the plots that take ResFactor into account are comparable with the plots of the grid-based algorithms.

Crossings (see Figure 13): since *Lattice* and *Visibility* use the same planarizer, the drawings produced by them have the same number of edge-crossings. *Dot* has the best performance among the four. The difference between the performances of the layering-based algorithms and of the grid-based algorithms reduces considerably for the Pert DAGs. Also, the slope of the plots is steeper for the Pert DAGs. This reflects the fact that the Pert DAGs are in general denser than the North DAGs and that the number of edge-crossings tends to increase with the ratio of the number of edges to the number of vertices.

ScreenRatio (see Figure 13): *Lattice* seems to be the algorithm of choice with respect to this quality measure. All the algorithms have a better performance for the Pert DAGs. This seems to be a consequence of the relative density of the Pert DAGs; the drawings tend to spread in both the x- and y-dimension.

TotalBends (see Figure 14): the performance of *Visibility* is unsatisfactory. For the North DAGs, the plots of the other three algorithms grow slowly for edge numbers up to 65. After that, *Dot* is clearly the best. The experimentation with the Pert DAGs produced a surprising result. Namely, *Lattice* outperforms the layering-based algorithms while *Visibility* has still the worst behavior. As for quality measure Crossings, the slope of the plots is steeper for the Pert DAGs. The behavior of *Lattice*, however, seems to be quite independent from the density of the input DAG, at least for DAGs with up to 75 edges.

MaxBends (see Figure 14): quite interestingly, the plots relative to the Pert DAGs grow linearly for all four algorithms. While *Dot* has the best performance for the North DAGs, *Lattice* is the best for the Pert DAGs. The overall performance of the algorithms is much better for the North DAGs than for the Pert DAGs. Again, this seems to be due to the fact that the North DAGs are in general sparser.

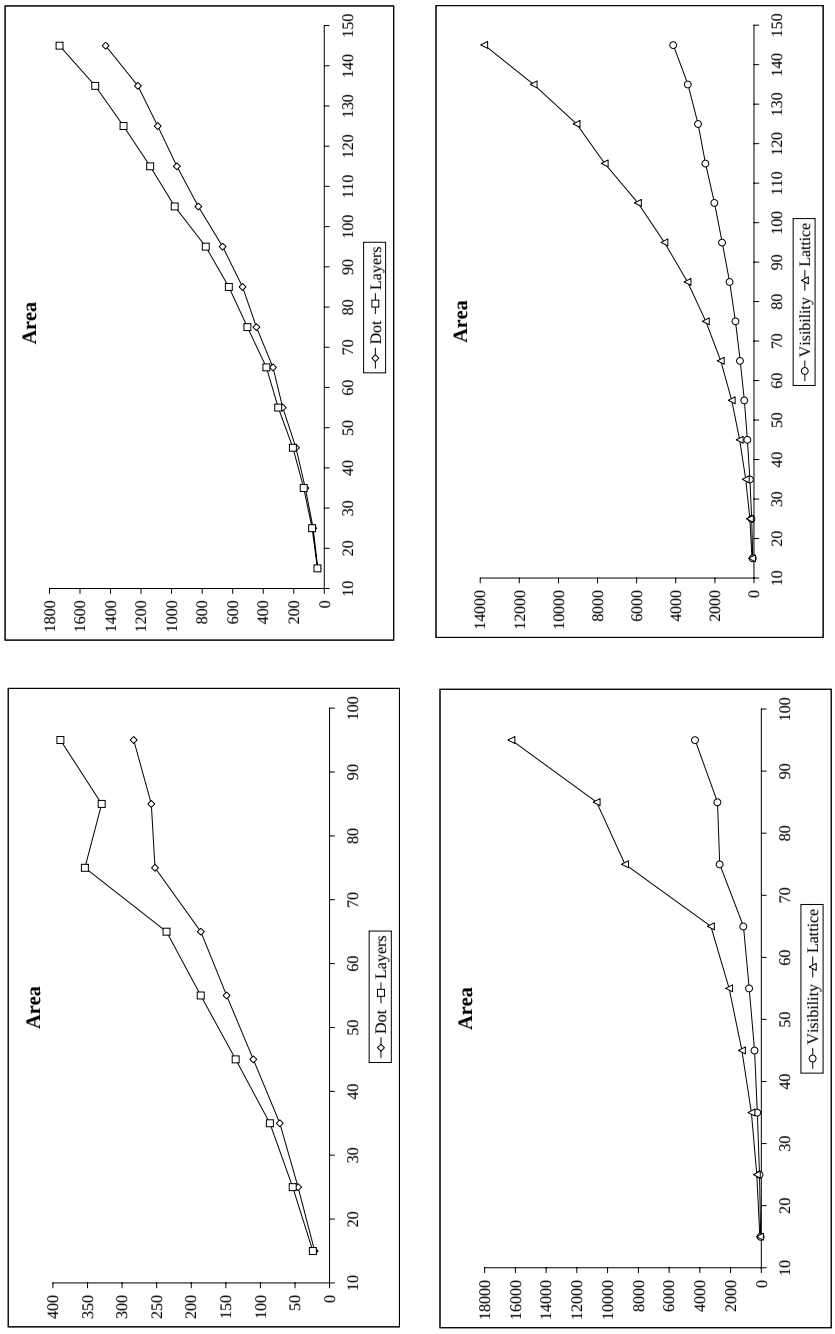

Figure 11. Comparison charts for Area: the x-axes indicate the number of edges.

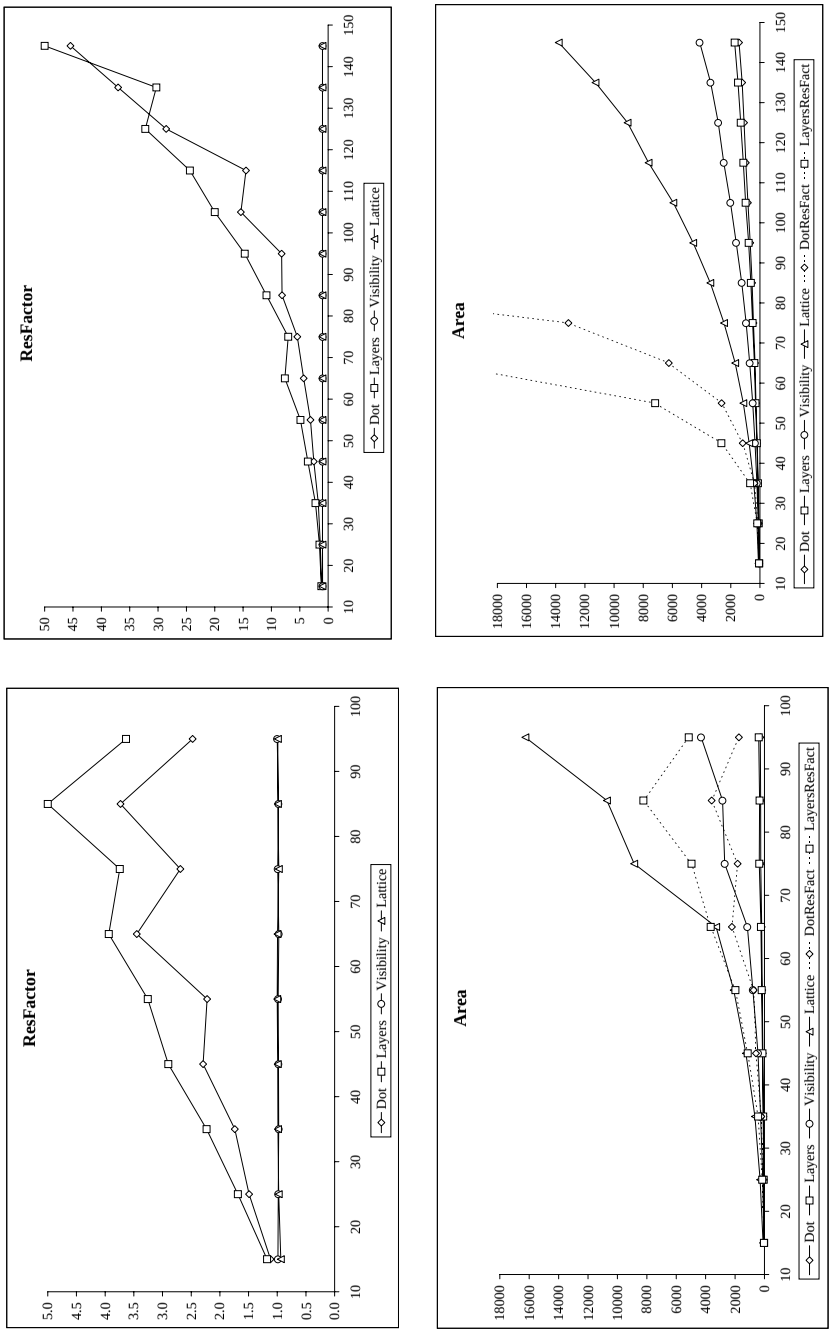

Figure 12. Comparison charts for ResFactor and Area: the x-axes indicate the number of edges.

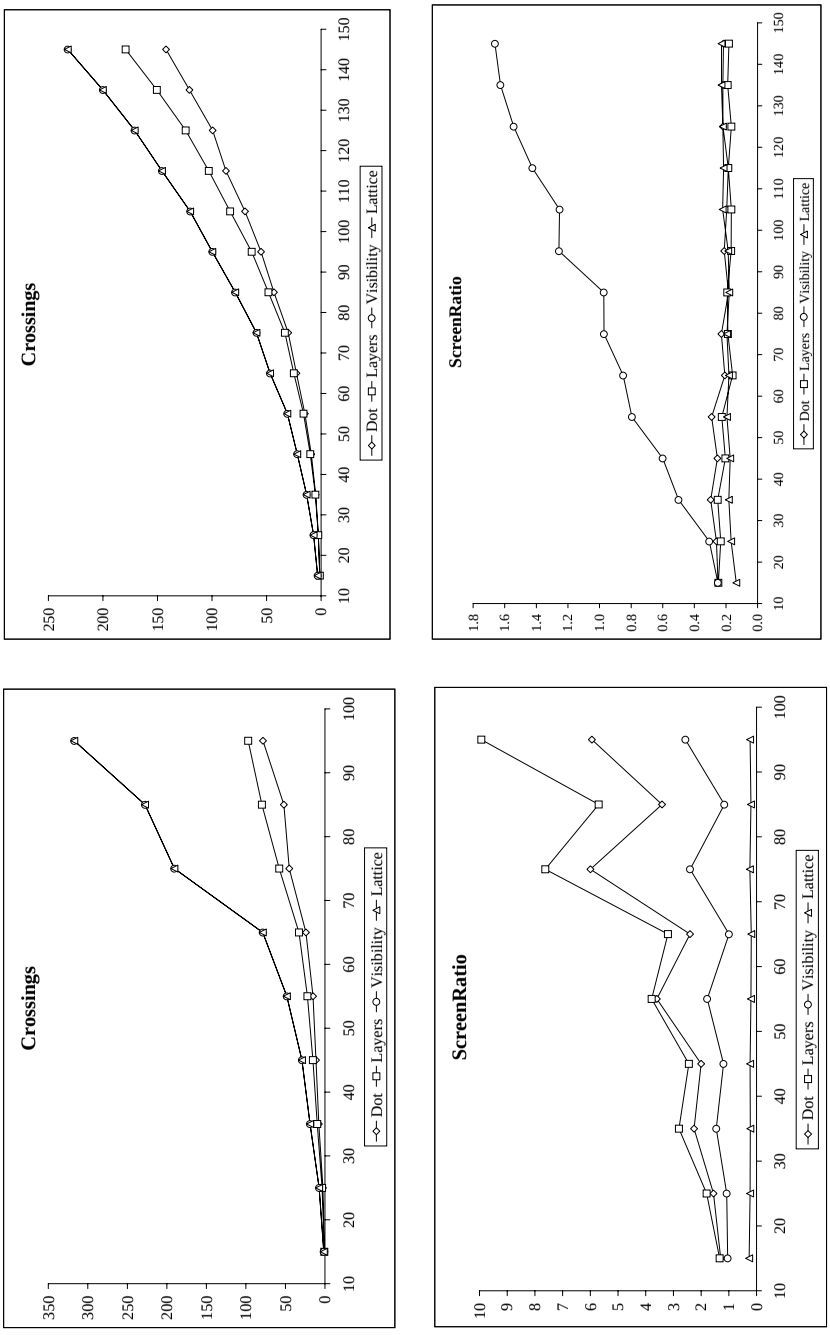

Figure 13. Comparison charts for Crossings and ScreenRatio: the x-axes indicate the number of edges.

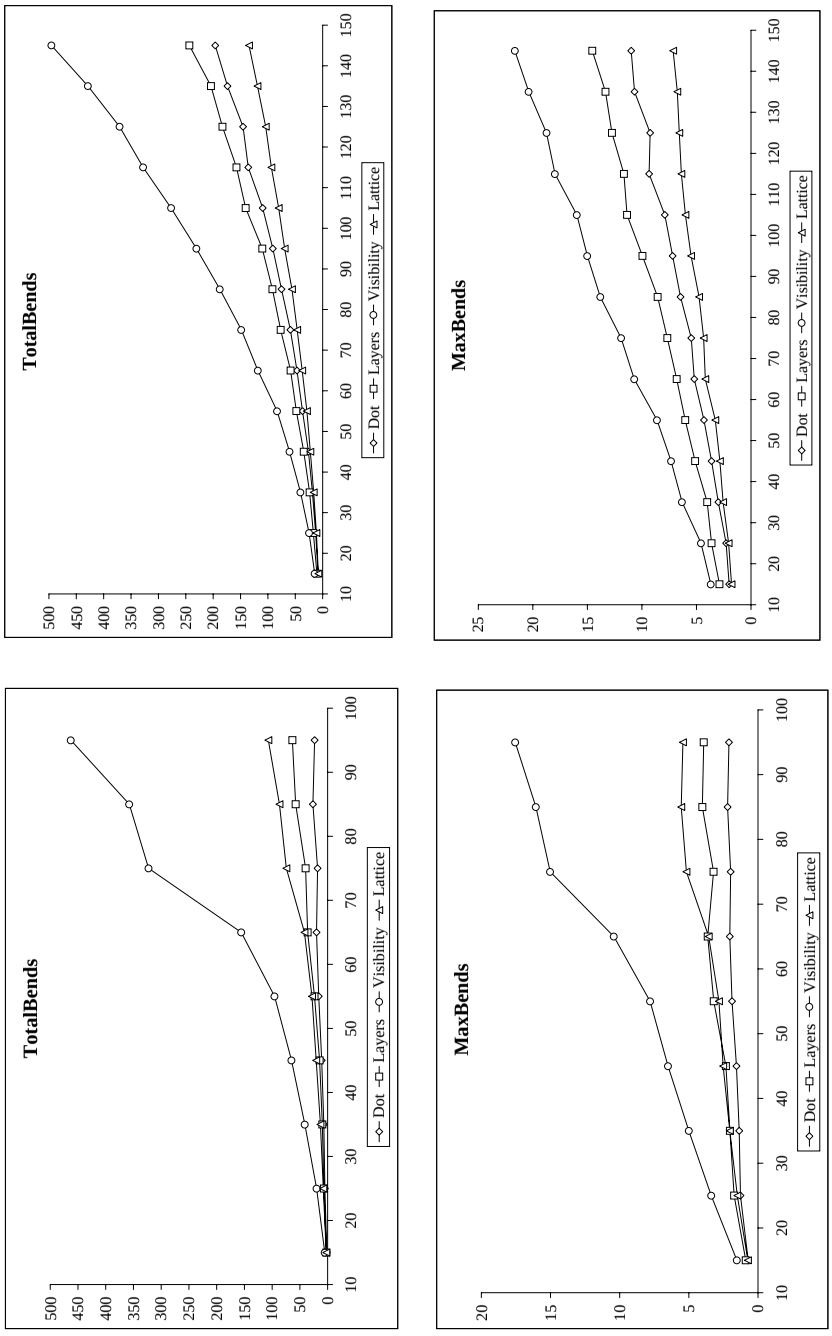

Figure 14. Comparison charts for TotalBends and MaxBends: the x-axes indicate the number of edges.

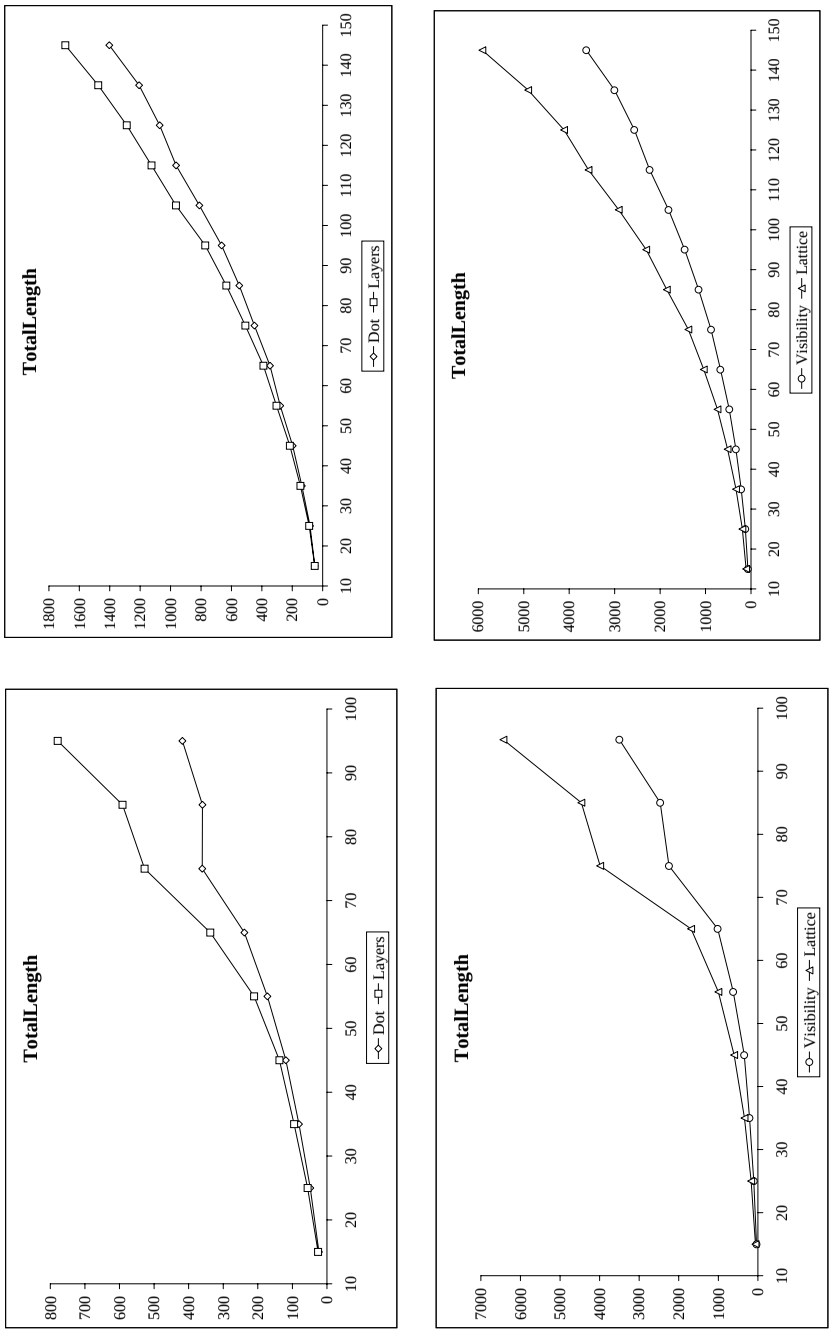

Figure 15. Comparison charts for TotalLength: the x-axes indicate the number of edges.

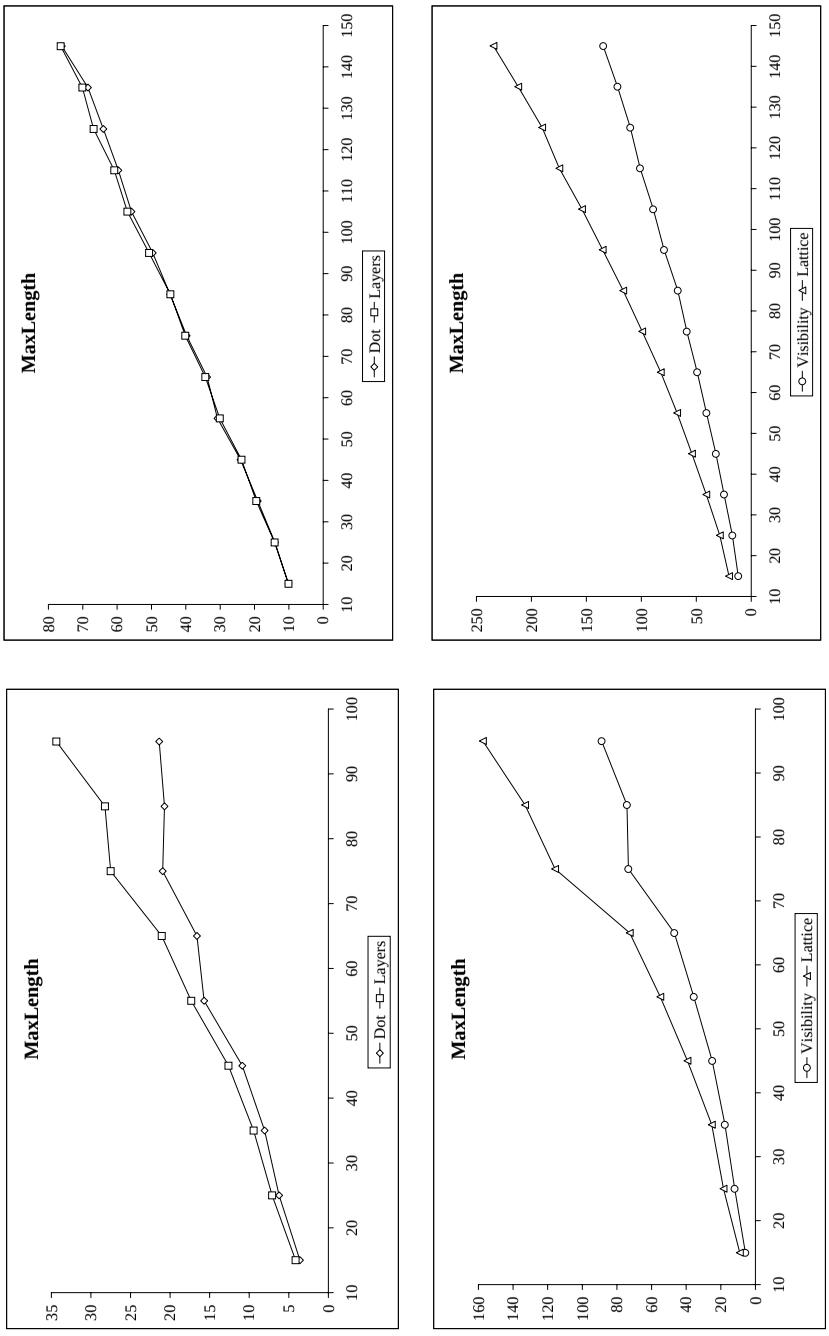

Figure 16. Comparison charts for MaxLength: the x-axes indicate the number of edges.

TotalLength and MaxLength (see Figures 15 and 16): these two quality measures are dependent on ResFactor; therefore, the layering-based and the grid-based algorithms were compared separately. *Dot* performs better than *Layers* and *Visibility* performs better than *Lattice* for both the North DAGs and the Pert DAGs.

In conclusion, the four algorithms under evaluation exhibit various trade-offs with respect to the quality measures, and none of them clearly outperforms the others. The sometimes surprising findings of the study include:

- Some algorithms construct very compact drawings at the expense of a relaxed resolution rule that does not consider crossing-crossing and vertex-crossing distances. Other algorithms produce drawings that distribute vertices and crossings with great regularity at the expenses of a larger area requirement.
- Concerning bends, an algorithm with good theoretical worst-case bounds (*Visibility*) performs in practice worse than algorithms for which no theoretical bounds are available.
- Concerning crossings, grid-based algorithms tend to perform worse than layering-based algorithms, where part of the geometry of the drawing is decided at the very first step.
- The performance of a drawing algorithm on planar DAGs is not a good predictor of the performance of the same algorithm on nonplanar DAGs.
- Algorithms with a topological foundation tend to distribute the bends and the lengths of the edges more evenly.

4.2.4. Cross-fertilization of layering- and grid-based algorithms

The analysis of the experimental results of *Dot*, *Layers*, *Visibility*, and *Lattice* clearly shows that the layering-based algorithms (*Dot* and *Layers*) produce drawings with fewer crossings than the grid-based algorithms (*Visibility* and *Lattice*). This indicates that the crossing reduction step of the layering-based algorithms is more effective than the simple planarization strategy [37] used in *Visibility* and *Lattice*. On the other hand, *Visibility* and *Lattice* perform well with respect to other quality measures (see Section 4.2.3).

The above considerations suggest the development of a hybrid strategy that substitutes the original planarization step of *Visibility* and *Lattice* with the crossing reduction step of *Layers*. More specifically, one may first execute the crossing reduction step of *Layers* and then visit the resulting drawing, replacing each crossing with a fictitious vertex. This planarizes the input graph, and the remaining algorithmic steps of *Visibility* and *Lattice* can be executed. In the study, the new drawing algorithms so obtained are called *VisibilityLayers* and *LatticeLayers*, respectively.

The charts in Figure 17 compare algorithms *Dot*, *Layers*, *Visibility*, *Lattice*, *VisibilityLayers*, and *LatticeLayers* with respect to quality measures Area and TotalBends. As in Section 4.2.3, the left column of the figure contains the charts for the North DAGs, and the right column contains those for the Pert DAGs; the x-axis of each chart indicates the number of edges.

Algorithms *VisibilityLayers* and *LatticeLayers* always perform better than their 'parent algorithms' *Visibility* and *Lattice*, respectively. In particular, the authors observe the following:

Area (see Figure 17): the improvement of *VisibilityLayers* and *LatticeLayers* over *Visibility* and *Lattice* is especially significant for the North DAGs, where it ranges between 30% and 50%.

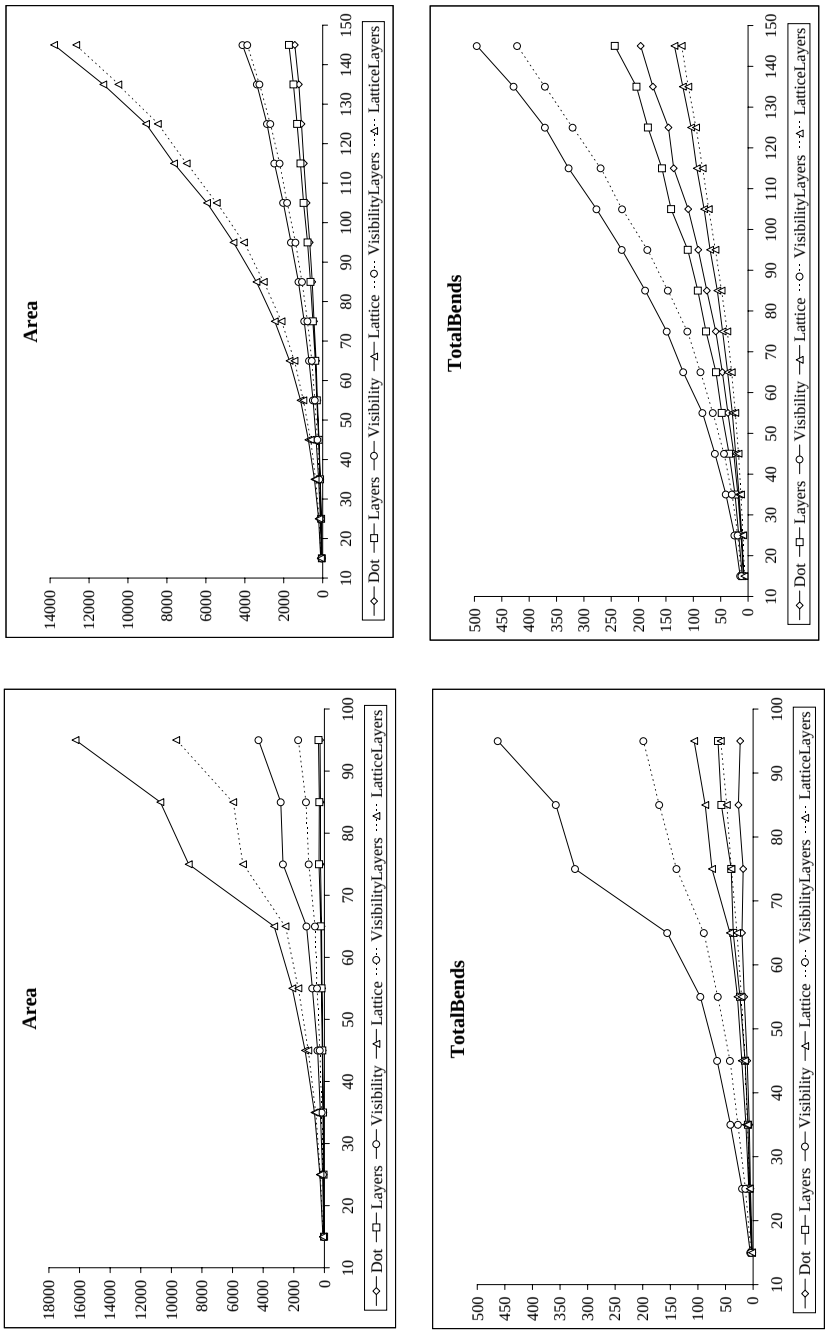

Figure 17. Comparison charts for Area and TotalBends: the x-axes indicate the number of edges.

TotalBends (see Figure 17): again, the improvement is especially significant (about 50%) for the North DAGs. Also, while *Layers* is always better than *Lattice*, *LatticeLayers* is slightly better than *Layers* for the North DAGs with more than 70 edges.

The analysis of the charts for the other quality measures shows similar trends, indicating that the new hybrid strategy performs quite well in practice. In general, one can observe that the performance of the grid-based algorithms (*Visibility* and *Lattice*) is strongly influenced by the number of crossings introduced in the planarization step.

5. FORCE-DIRECTED AND RANDOMIZED ALGORITHMS

One of the most attractive aspects of graph drawing algorithms based on force-directed methods is the variety of possible applications, since they can be used for drawing general undirected graphs. As a consequence, several algorithms and tools for graph drawing based on these methods have been developed and experimented during the years (see [1,11] for references). Since an exhaustive analysis of all these algorithms exceeds the scope of this paper, we restrict our attention to some of them that, in our opinion, are among the most relevant. We also assume that the reader is familiar with algorithmic techniques such as simulated annealing and randomization.

A milestone of the force-directed graph drawing literature is the work by Eades [64]. Eades' idea is to model a graph G as a mechanical system where the vertices of G are steel rings and the edges of G are springs that connect pairs of rings. In order to compute a drawing of G, the rings are placed in some initial layout and let go until the spring forces move the system to a local minimum of energy. Clearly, different definitions of the repulsive and attractive forces that characterize the physical model of G give rise to different drawings of G. For example, Eades' implementation of the physical model does not reflect Hooke's law for the forces, but obeys a logarithmic function for attractive forces; also, while repulsive forces act between every pair of vertices, attractive forces are calculated only between pairs of adjacent vertices. Most algorithms that are considered in the rest of this section adopt the physical model by Eades and propose variants of the forces definition.

5.1. The magnetic spring model

Sugiyama and Misue [24] propose a variant of Eades' model that is particularly suited for drawing directed graphs, trees, and graphs with more than one type of edges (e.g., graphs with both directed and undirected edges). The main innovations are the concept of magnetic springs and the use of a magnetic field acting on these springs to control the orientation of the edges in the drawing. In particular, edges can be modeled with three types of springs: non-magnetic springs, unidirectional magnetic springs, and bidirectional magnetic springs. Various magnetic fields are considered: three standard magnetic fields, namely, parallel, polar, and concentric, and two compound magnetic fields, namely, orthogonal and polar-concentric. In the same paper, the authors also present an experimental study on the magnetic spring model. In what follows, we denote the algorithm of Sugiyama and Misue by *MagneticSpring*.

5.1.1. Experimental setting

5.1.1.1. Quality measures analyzed. Two of the quality measures analyzed, namely, Crossings and AvgLength, have already been described in Sections 2.1.2.1 and 3.3.1.1. The others are:

VarLength: variance of the edge length;

OrientationErrors: percentage of the edges whose orientation does not approximately conform to the direction of the magnetic field;

AvgOrientationAngle: average angle between the orientation of an edge and the direction of the magnetic field;

VarOrientationAngle: variance of the angle between the orientation of an edge and the direction of the magnetic field;

MinVertexDistance: minimum distance between a vertex and its nonadjacent vertices, intended as a measure of the density of the vertices.

5.1.1.2. Test suite. The test suite consists of randomly generated graphs belonging to classes that are relevant for graph drawing applications, including rooted trees (RTs), directed acyclic graphs (DAGs), and directed cyclic graphs (DCGs). For each class, 30 graphs are considered, whose number of vertices is either 20 or 40, and whose average vertex degree varies between 2 and 3. For each type of magnetic field, *MagneticSpring* has been repeatedly run on the test graphs with increasing values of the magnetic field strength.

5.1.2. Analysis of the experimental results

By analyzing the experimental results and the produced drawings, the authors observe that:

- A parallel magnetic field is particularly effective for drawing trees almost without crossings (i.e., small values of Crossings), and for drawing directed acyclic graphs upward or directed cyclic graphs with a small number of feedback edges (i.e., in both cases, small values of OrientationErrors); the algorithm performs well for large values of the magnetic field strength.
- An orthogonal magnetic field can be used for h-v drawings of rooted binary trees, but the drawing may contain a small number of edge-crossings; the algorithm performs well for all values of the magnetic field strength.
- A polar magnetic field can be used for drawings of bipartite graphs with few edge-crossings, placing the vertices on two concentric discs.
- A concentric magnetic field can be used for highlighting the existence of cycles, but, again, the drawing may contain a small number of edge-crossings; the algorithm performs well for intermediate values of the magnetic field strength.
- Symmetries in the graphs are often revealed in the produced drawings.

In conclusion, *MagneticSpring* seems particularly effective with respect to quality measures such as Crossings, OrientationErrors, AvgOrientationAngle, and VarOrientationAngle, while it is less effective for keeping the edge lengths uniform or for evenly distributing vertices.

5.2. Comparing five force-directed and randomized algorithms

An extensive experimental comparison of five graph drawing algorithms based on force-directed and randomized methods is described in the work by Brandenburg *et al.* [23].

5.2.1. The drawing algorithms under evaluation

Fruchterman and Reingold [40] have proposed a variant of Eades' approach, in which attractive forces take into account the optimal distance between vertices in the drawing defined as a function of the number of vertices in the graph and the size of the drawing window. In what follows we denote the algorithm of Fruchterman and Reingold by *Spring-FR*.

In the model by Kamada and Kawai [41], the forces are chosen so that the Euclidean distance between any two vertices in the drawing is proportional to the graph-theoretic distance between the corresponding vertices in the graph. In what follows we denote the algorithm of Kamada and Kawai by *Spring-KK*.

While the above mentioned methods are based on continuous functions of the locations of the vertices, Davidson and Harel [42] minimize a function that is a linear combination of energy functions, each defined to optimize a certain aesthetic criterion, such as uniform distribution of vertices, uniform distribution of edge lengths, number of edge-crossings, and distance between vertices and edges. Users can rank the aesthetic criteria that they want to be satisfied in the drawing by appropriately choosing the values of the coefficients in the linear combination. An optimal solution is searched by means of a simulated annealing technique. In what follows we denote the algorithm of Davidson and Harel by *SimulatedAnnealing-DH*.

Tunkelang [43] elaborates on the cost function of Davidson and Harel and shows a method for finding a local minimum. The vertices are inserted one at a time in a given order (e.g., breadth-first search from the center of the graph). At each insertion of a vertex v, the algorithm checks some candidate positions for v, chooses the one that corresponds to a minimum of energy, and performs some fine tuning. The choice of the candidate positions depends on a quality parameter, whose value is chosen by the user. In what follows we denote the algorithm of Tunkelang by *SimulatedAnnealing-T*.

Frick *et al.* [44] have devised a randomized adaptive algorithm that further refines the approach by Davidson and Harel by means of a cooling schedule based on a local temperature, associated with each vertex, and a global temperature defined as the average of the local temperatures. Local temperatures rise if the algorithm determines that a vertex is far from its final destination. The lower the global temperature, the stabler the drawing. In what follows we denote the algorithm of Frick, Ludwig and Mehldau by *Randomized-GEM*.

For the experimental study, the authors have used the implementations of *Spring-FR*, *Spring-KK*, *SimulatedAnnealing-DH*, *SimulatedAnnealing-T* and *Randomized-GEM* available in *GraphEd* [84], an extensible editor for graphs and graph grammars.

5.2.2. Experimental setting

5.2.2.1. Quality measures analyzed. The quality measures analyzed are some of those described in Section 2.1.2.1, namely, Crossings, UnifLength and Time, plus the following one:

LengthRatio: ratio of the maximum to minimum edge lengths.

5.2.2.2. Test suite. The set of graphs that the authors use for testing can be split into two subsets. The *mixed graphs* are 59 graphs taken from the papers by Davidson and Harel [42] and by Fruchterman and Reingold [40]. The *complete graphs* consists of all complete graphs from K_5 to K_{24}. For each graph, the algorithms have been executed several times and the data of the best, the worst, and the average computation have been collected. The authors observe that the corresponding results are not very different, except for the number of crossings and when the number of iterations is too small.

5.2.2.3. Computing equipment. The experiments were performed on a Sun Sparc-10 workstation.

5.2.3. Analysis of the experimental results

Figures 18 and 19 show the average values of the quality measures. The x-axis of each chart indicates the size of the graph (either the number of vertices for the complete graphs, or the number of vertices plus the number of edges for the mixed graphs). The average values of the quality measures are computed over each group of graphs with a given size. Figure 20 shows different behaviors of *SimulatedAnnealing-DH* on the complete graphs, depending on whether the number of edge-crossings is optimized (dashed plots) or not (full plots).

From the charts and from the visual analysis of the produced drawings, the authors confirm in practice the behavior of the algorithms as described in the original papers. No algorithm, among those evaluated, is elected as the best one for all graphs. On the contrary, they recommend trying several methods and choose the one that produces the most aesthetically pleasing drawing. They also observe that their implementation of *Spring-KK* and *Randomized-GEM* outperforms the implementations of the other algorithms in terms of running time. On relatively small input instances (graphs with less than 60 vertices), also *Spring-FR* is relatively fast. On the other hand, *SimulatedAnnealing-DH* and *SimulatedAnnealing-T* are more flexible than the other algorithms, even though they may be slower. Hence, if time is not critical, one can adjust the parameters of *SimulatedAnnealing-DH* and *SimulatedAnnealing-T* to obtain pleasing drawings. The authors also observe that for *SimulatedAnnealing-DH* it is often difficult to satisfy many aesthetic requirements at the same time. For example, minimizing the number of crossings and optimizing the distance between vertices and edges may lead to drawings where display of symmetries, uniformity of edge lengths, and uniform distribution of vertices are not satisfying. As for *Spring-KK*, the produced drawings have low LengthRatio and UnifLength.

Another interesting observation is that *Spring-FR*, *Spring-KK*, *SimulatedAnnealing-DH* (without edge-crossing optimization), and *Randomized-GEM* produce drawings that are quite similar to each other and display well the symmetries for complete or almost complete graphs. Conversely, the drawings produced by *SimulatedAnnealing-T* tend to be different from those produced by the other algorithms. Hence, *SimulatedAnnealing-T* can be considered as an option to try when all other algorithms fail to match the aesthetic requirements. Also, *SimulatedAnnealing-T* does not perform very well on displaying symmetries, but seems to be particularly effective for the layout of grids and net structures.

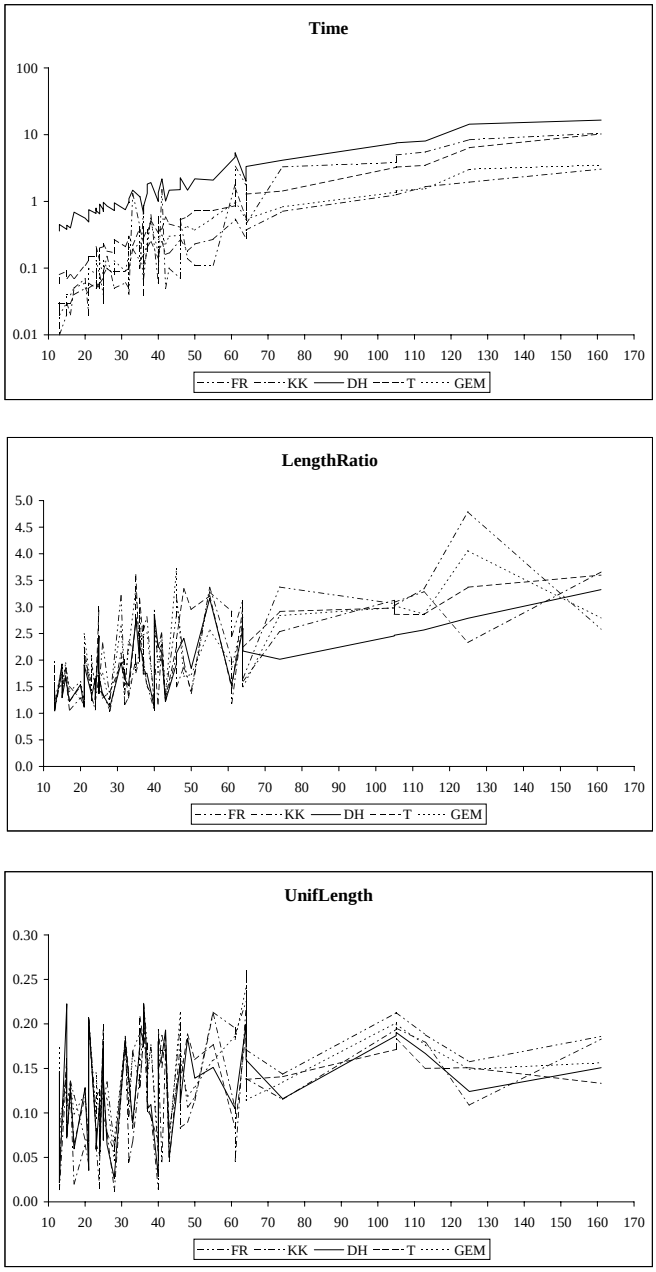

Figure 18. Comparison charts for Time (seconds), LengthRatio, and UnifLength relative to the mixed graphs: the x-axes indicate the number of vertices plus the number of edges. (Data courtesy of Franz J. Brandenburg, Michael Himsolt and Christoph Rohrer.)

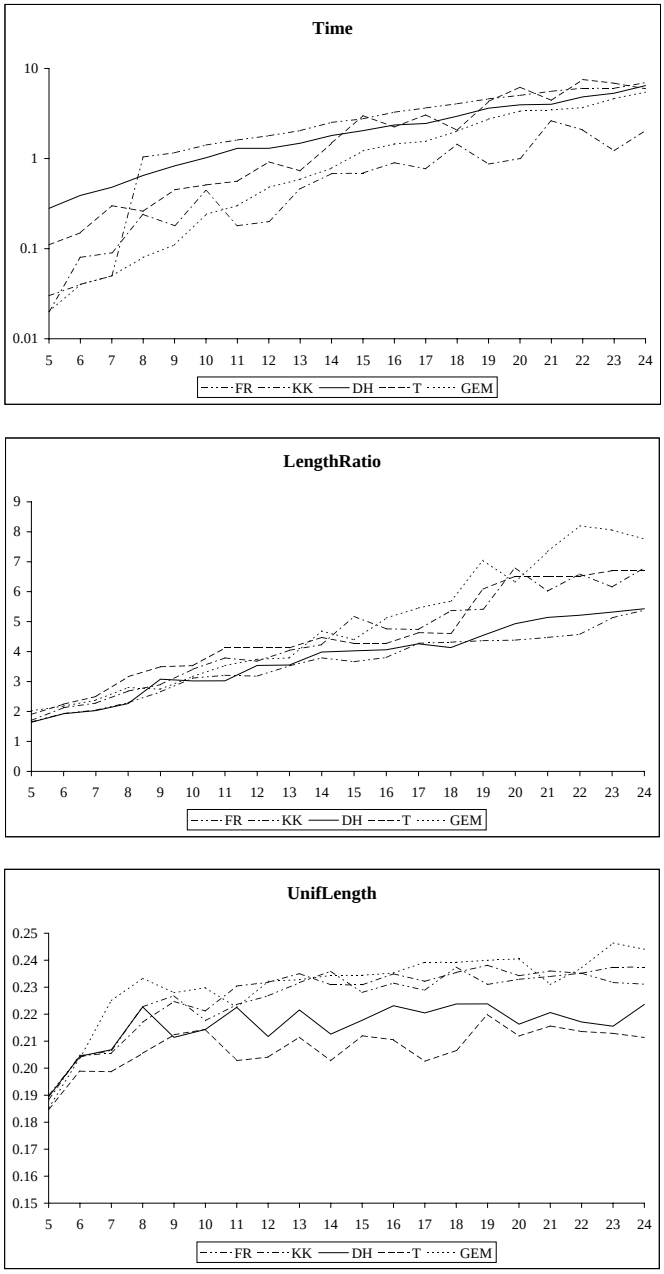

Figure 19. Comparison charts for Time (seconds), LengthRatio, and UnifLength relative to the complete graphs: the x-axes indicate the number of vertices. (Data courtesy of Franz J. Brandenburg, Michael Himsolt and Christoph Rohrer.)

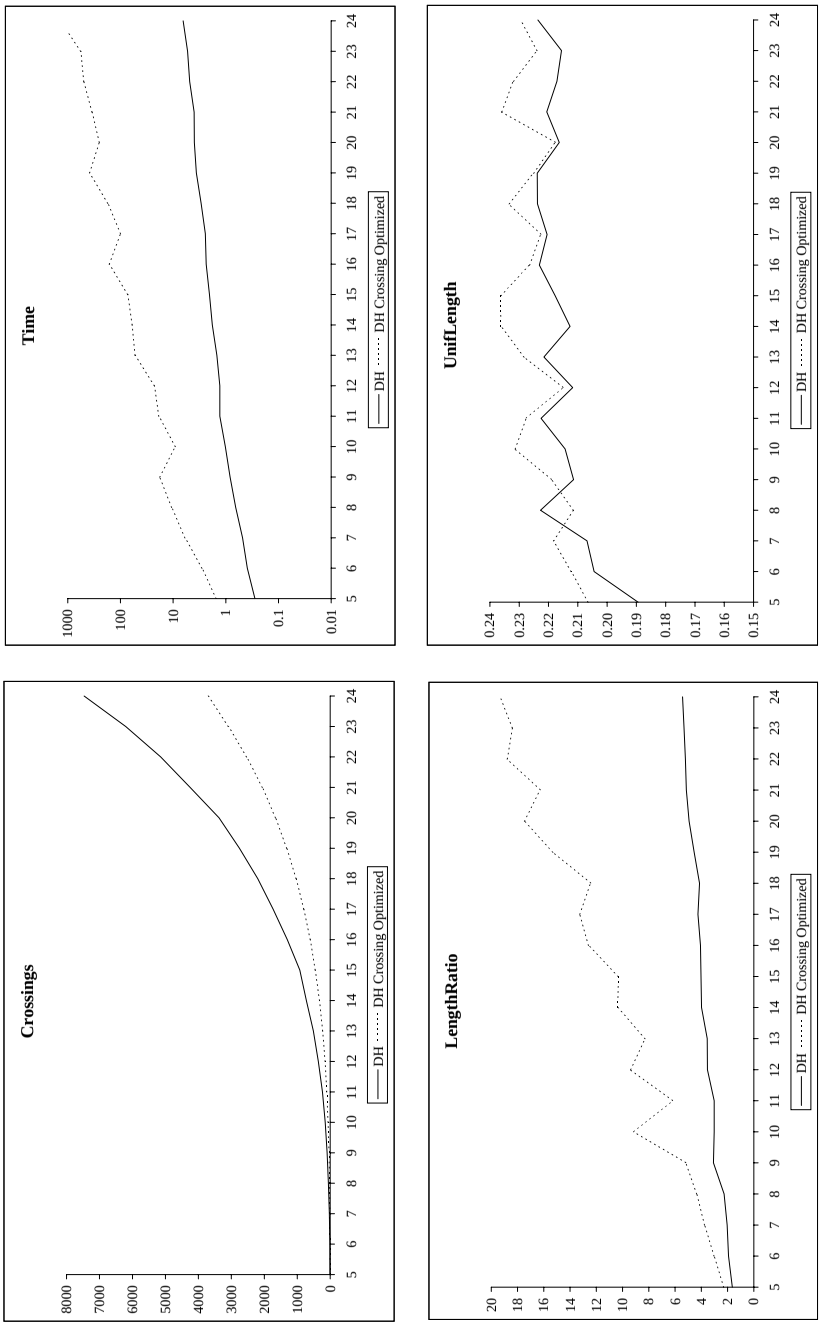

Figure 20. Comparison charts for Crossings, Time (seconds), LengthRatio, and UnifLength when *SimulatedAnnealing-DH* is executed on the complete graphs with or without edge-crossing optimization: the x-axes indicate the number of vertices. (Data courtesy of Franz J. Brandenburg, Michael Himsolt and Christoph Rohrer.)

Table IV. The quality measures analyzed for each algorithm under evaluation.

	Bend-Stretch	Column	Giotto	Pair	No-Change	Relative-Coordinates	Barycentric	Median	Stochastic	GreedyInsert	GreedySwitch	Split	Assignment	Branch & Cut	Branch & Bound (Branch & Cut)	Iterative (Barycentric)	Iterative (Branch & Cut)	Dot	Layers	Visibility	Lattice	MagneticSpring	Spring-FR	Spring-KK	SimulatedAnnealing-DH	SimulatedAnnealing-T	Randomized-GEM
Crossings	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
TotalBends	•	•	•	•	•	•												•	•	•	•						
MaxBends	•	•	•	•														•	•	•	•						
UnifBends	•	•	•	•																							
ResFactor																		•	•	•	•						
Area	•	•	•	•	•	•												•	•	•	•						
Area (ResFactor)																		•	•	•	•						
TotalLength	•	•	•	•														•	•	•	•						
TotalLength (ResFactor)																		•	•	•	•						
MaxLength	•	•	•	•	•	•												•	•	•	•						
MaxLength (ResFactor)																		•	•	•	•						
AvgLength					•	•																•					
VarLength																						•					
UnifLength	•	•	•	•																			•	•	•	•	•
LengthRatio																							•	•	•	•	•
ScreenRatio	•	•	•	•														•	•	•	•						
AspectRatio					•	•																					
OrientationErrors																						•					
AvgOrientationAngle																						•					
VarOrientationAngle																						•					
MinVertexDistance																						•					
NewRows					•	•																					
NewColumns					•	•																					
NewBends					•	•																					
Time	•	•	•	•			•	•	•	•	•	•	•	•	•	•	•						•	•	•	•	•

6. CONCLUSIONS

The results of the experimental studies described in this paper suggest the following considerations:

- Graph drawing has a tradition of combining theoretical and applied work, and more than ten years of research have produced a complex landscape. Experimental studies are essential to assess the practical performance of graph drawing algorithms in real-life applications.
- In general, the algorithms under evaluation exhibit various trade-offs with respect to the quality measures analyzed, and, in general, none of them clearly outperforms the others. When this happens, as for *Giotto* among the algorithms for orthogonal drawings, the benefits are paid in terms of a substantially higher running time.
- The theoretical analysis of graph drawing algorithms for special classes of graphs (e.g., planar, biconnected, maximum degree 4, etc) is insufficient to predict the behavior of algorithms for general graphs derived from them (see, e.g., *Bend-Stretch*, *Column*, *Visibility*, and *Lattice*).

The paper is summarized in Table IV, where, for each algorithm under evaluation, we indicate the quality measures that have been analyzed.

ACKNOWLEDGEMENTS

We would like to thank the authors of the described studies for providing and allowing us to use their experimental data. In particular, our thanks go to Achilleas Papakostas, Janet M. Six and Ioannis G. Tollis for Section 3, to Michael Jünger and Petra Mutzel for Section 4.1, and to Franz J. Brandenburg, Michael Himsolt and Christoph Rohrer for Section 5.2.

We would also like to thank the anonymous referees for useful comments on how to improve the presentation of the paper.

REFERENCES

1. Di Battista G, Eades P, Tamassia R, Tollis IG. Algorithms for drawing graphs: An annotated bibliography. *Comput. Geom. Theory Appl.* 1994; **4**(5):235–282.
2. Brandenburg FJ (ed.). *Graph Drawing (Proceedings of GD '95)* (*Lecture Notes in Computer Science*, vol. 1027). Springer-Verlag, 1996.
3. Di Battista G (ed.). *Graph Drawing (Proceedings of GD '97)* (*Lecture Notes in Computer Science*, vol. 1353). Springer-Verlag, 1997.
4. Di Battista G, Eades P, de Fraysseix H, Rosenstiehl P, Tamassia R (eds.). *Graph Drawing '93 (Proceedings of the ALCOM International Workshop on Graph Drawing)*, 1993. http://www.cs.brown.edu/people/rt/gd-93.html.
5. North S (ed.). *Graph Drawing (Proceedings of GD '96)* (*Lecture Notes in Computer Science*, vol. 1190). Springer-Verlag, 1997.
6. Tamassia R, Tollis IG (eds.). *Graph Drawing (Proceedings of GD '94)* (*Lecture Notes in Computer Science*, vol. 894). Springer-Verlag, 1995.
7. Whitesides SH (ed.). *Graph Drawing (Proceedings of GD '98)* (*Lecture Notes in Computer Science*, vol. 1547) Springer-Verlag, 1998.
8. Cruz IF, Eades P (eds.). Special issue on graph visualization. *J. Visual Lang. Comput.* 1995; **6**(3).
9. Di Battista G, Tamassia R (eds.). Special issue on graph drawing. *Algorithmica* 1996; **16**(1).
10. Di Battista G, Tamassia R (eds.). Special issue on geometric representations of graphs. *Comput. Geom. Theory Appl.* 1998; **9**(1–2).
11. Di Battista G, Eades P, Tamassia R, Tollis IG. *Graph Drawing*. Prentice Hall: Upper Saddle River, NJ, 1999.

12. Tamassia R, Di Battista G, Batini C. Automatic graph drawing and readability of diagrams. *IEEE Trans. Syst. Man Cybern.* 1988; **SMC-18**(1):61–79.
13. Purchase H. Which aesthetic has the greatest effect on human understanding? *Graph Drawing (Proceedings of GD '97)* (*Lecture Notes in Computer Science*, vol. 1353), Di Battista G (ed.). Springer-Verlag, 1997; 248–261.
14. Purchase HC, Cohen RF, James MI. An experimental study of the basis for graph drawing algorithms. *ACM J. Experim. Algorithmics* 1997; **2**(4). http://www.jea.acm.org/1997/PurchaseDrawing/.
15. Johnson DS. A theoretician's guide to the experimental analysis of algorithms. *Preliminary, partial draft*, 1996; http://www.research.att.com/~dsj/papers/exper.ps.
16. Himsolt M. Comparing and evaluating layout algorithms within GraphEd. *J. Visual Lang. Comput.* 1995; **6**(3):255–273. Special Issue on Graph Visualization, Cruz IF, Eades P (eds.).
17. Di Battista G, Garg A, Liotta G, Tamassia R, Tassinari E, Vargiu F. An experimental comparison of four graph drawing algorithms. *Comput. Geom. Theory Appl.* 1997; **7**(5–6):303–325.
18. Papakostas A, Six JM, Tollis IG. Experimental and theoretical results in interactive orthogonal graph drawing. *Graph Drawing (Proceedings of GD '96)* (*Lecture Notes in Computer Science*, vol. 1190), North S (ed.). Springer-Verlag, 1997; 371–386.
19. Papakostas A, Six JM, Tollis IG. Interactive orthogonal graph drawing: An experimental study. *Technical Report UTDCS-3-99*, Dept. Comput. Sci., Univ. Texas Dallas, May 1999.
20. Di Battista G, Garg A, Liotta G, Parise A, Tamassia R, Tassinari E, Vargiu F, Vismara L. Drawing directed acyclic graphs: An experimental study. *Graph Drawing (Proceedings of GD '96)* (*Lecture Notes in Computer Science*, vol. 1190), North S (ed.). Springer-Verlag, 1997; 76–91.
21. Di Battista G, Garg A, Liotta G, Parise A, Tamassia R, Tassinari E, Vargiu F, Vismara L. Drawing directed acyclic graphs: An experimental study. *Int. J. Comput. Geom. Appl.* to appear.
22. Jünger J, Mutzel P. 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. *J. Graph Algorithms Appl.* 1997; **1**(1):1–25. http://www.cs.brown.edu/publications/jgaa/accepted/97/JuengerMutzel97.1.1.ps.gz.
23. Brandenburg FJ, Himsolt M, Rohrer C. An experimental comparison of force-directed and randomized graph drawing algorithms. *Graph Drawing (Proceedings of GD '95)* (*Lecture Notes in Computer Science*, vol. 1027), Brandenburg FJ (ed.). Springer-Verlag, 1996; 76–87.
24. Sugiyama K, Misue K. Graph drawing by the magnetic spring model. *J. Visual Lang. Comput.* 1995; **6**(3):217–231. Special Issue on Graph Visualization, Cruz IF, Eades P (eds.).
25. Tamassia R, Tollis IG. Planar grid embedding in linear time. *IEEE Trans. Circuits Syst.* 1989; **CAS-36**(9):1230–1234.
26. Biedl TC, Kant G. A better heuristic for orthogonal graph drawings. *Comput. Geom. Theory Appl.* 1998; **9**(3):159–180.
27. Tamassia R. On embedding a graph in the grid with the minimum number of bends. *SIAM J. Comput.* 1987; **16**(3):421–444.
28. Papakostas A, Tollis IG. Improved algorithms and bounds for orthogonal drawings. *Graph Drawing (Proceedings of GD '94)* (*Lecture Notes in Computer Science*, vol. 894), Tamassia R, Tollis IG (eds.). Springer-Verlag, 1995; 40–51.
29. Papakostas A, Tollis IG. Algorithms for area-efficient orthogonal drawings. *Comput. Geom. Theory Appl.* 1998; **9**(1–2):83–110. Special Issue on Geometric Representations of Graphs, Di Battista G, Tamassia R (eds.).
30. Papakostas A, Tollis IG. Interactive orthogonal graph drawing. *IEEE Trans. Comput.* 1998; **C-47**(11):1297–1309.
31. Sugiyama K, Tagawa S, Toda M. Methods for visual understanding of hierarchical systems. *IEEE Trans. Syst. Man Cybern.* 1981; **SMC-11**(2):109–125.
32. Eades P, Wormald NC. Edge crossings in drawings of bipartite graphs. *Algorithmica* 1994: 11(4):379–403.
33. Dresbach S. A new heuristic layout algorithm for DAGs. *Operations Research Proceedings 1994*, Derigs U, Bachem A, Drexl A (eds.). Springer-Verlag, 1994; 121–126.
34. Eades P, Kelly D. Heuristics for reducing crossings in 2-layered networks. *Ars Combin.* 1986; **21.A**:89–98.
35. Catarci T. The assignment heuristic for crossing reduction. *IEEE Trans. Syst. Man Cybern.* 1995; **SMC-25**(3):515–521.
36. Koutsofios E, North S. Drawing graphs with *dot*. 1993. ftp://ftp.research.att.com/dist/drawdag/dotdoc.ps.Z.
37. Di Battista G, Pietrosanti E, Tamassia R, Tollis IG. Automatic layout of PERT diagrams with XPERT. *Proceedings of the IEEE Workshop on Visual Languages (VL'89)*, 1989; 171–176.
38. Di Battista G, Tamassia R. Algorithms for plane representations of acyclic digraphs. *Theoret. Comput. Sci.* 1988; **61**(2,3):175–198.
39. Di Battista G, Tamassia R, Tollis IG. Area requirement and symmetry display of planar upward drawings. *Discrete Comput. Geom.* 1992; **7**(4):381–401.
40. Fruchterman T, Reingold E. Graph drawing by force-directed placement. *Softw.—Pract. Exp.* 1991; **21**(11):1129–1164.
41. Kamada T, Kawai S. An algorithm for drawing general undirected graphs. *Inform. Process. Lett.* 1989; **31**(1):7–15.
42. Davidson R, Harel D. Drawing graphs nicely using simulated annealing. *ACM Trans. Graph.* 1996; **15**(4):301–331.
43. Tunkelang D. A practical approach to drawing undirected graphs. *Technical Report CMU-CS-94-161*, School Comput. Sci., Carnegie Mellon Univ., June 1994.
44. Frick A, Ludwig A, Mehldau H. A fast adaptive layout algorithm for undirected graphs. *Graph Drawing (Proceedings of GD '94)*, (*Lecture Notes in Computer Science*, vol. 894). Tamassia R, Tollis IG (eds.). Springer-Verlag, 1995; 388–403.

45. Kant G. Algorithms for Drawing Planar Graphs. *PhD Thesis*, Dept. Comput. Sci., Univ. Utrecht, Utrecht, The Netherlands, 1993.
46. Trickey H. Drag: A graph drawing system. *Proceedings of the International Conference on Electronic Publishing*, 1988. Cambridge University Press, 171–182.
47. Di Battista G, Giammarco A, Santucci G, Tamassia R. The architecture of Diagram Server. *Proceedings of the IEEE Workshop on Visual Languages*, 1990; 60–65.
48. Di Battista G, Liotta G, Vargiu F. Diagram Server. *J. Visual Lang. Comput.* 1995; **6**(3):275–298. Special Issue on Graph Visualization, Cruz IF, Eades P (eds.).
49. Garey MR, Johnson DS. Crossing number is NP-complete. *SIAM J. Algebraic Discrete Methods* 1983; **4**(3):312–316.
50. Batini C, Furlani L, Nardelli E. What is a good diagram? A pragmatic approach. *Proceedings of the 4th International Conference on the Entity-Relationship Approach*, 1985; 312–319.
51. Batini C, Nardelli E, Tamassia R. A layout algorithm for data flow diagrams. *IEEE Trans. Softw. Eng.* 1986; **SE-12**(4):538–546.
52. Eades P, Lai W, Misue K, Sugiyama K. Preserving the mental map of a diagram. *Proceedings of the 1st International Conference on Computational Graphics and Visualisation Techniques*, 1991: 34–43.
53. Misue K, Eades P, Lai W, Sugiyama K. Layout adjustment and the mental map. *J. Visual Lang. Comput.* 1995; **6**(2):183–210.
54. Cohen RF, Di Battista G, Tamassia R, Tollis IG. Dynamic graph drawings: Trees, series-parallel digraphs, and planar st-digraphs. *SIAM J. Comput.* 1995; **24**(5):970–1001.
55. Moen S. Drawing dynamic trees. *IEEE Softw.* 1990; **7**(4):21–28.
56. Reingold E, Tilford J. Tidier drawing of trees. *IEEE Trans. Softw. Eng.* 1981; **SE-7**(2):223–228.
57. Newbery Paulish F, Tichy WF. EDGE: An extendible graph editor. *Softw.—Pract. Exp.* 1990; **20**(S1):63–88.
58. Bohringer K, Newbery Paulish F. Using constraints to achieve stability in automatic graph layout algorithms. *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 1990; 43–51.
59. North S. Incremental layout in DynaDAG. *Graph Drawing (Proceedings of GD '95)* (*Lecture Notes in Computer Science*, vol. 1027), Brandenburg FJ (ed.). Springer-Verlag, 1996; 409–418.
60. Miriyala K, Hornick SW, Tamassia R. An incremental approach to aesthetic graph layout. *Proceedings of the 6th International Workshop on Computer-Aided Software Engineering*. IEEE Computer Society, 1993; 297–308.
61. Biedl TC, Kaufmann M. Area-efficient static and incremental graph drawings. *Algorithms (Proceedings of ESA'97)* (*Lecture Notes in Computer Science*, vol. 1284), Burkard R, Woeginger G (eds.). Springer-Verlag, 1997; 37–52.
62. Bridgeman SS, Fanto J, Garg A, Tamassia R, Vismara L. InteractiveGiotto: An algorithm for interactive orthogonal graph drawing. *Graph Drawing (Proceedings of GD '97)* (*Lecture Notes in Computer Science*, vol. 1353), Di Battista G (ed.). Springer-Verlag, 1997; 303–308.
63. Brandes U, Wagner D. A bayesian paradigm for dynamic graph layout. *Graph Drawing (Proceedings of GD '97)* (*Lecture Notes in Computer Science*, vol. 1353), Di Battista G (ed.). Springer-Verlag, 1997; 236–247.
64. Eades P. A heuristic for graph drawing. *Congr. Numer.* 1984; **42**:149–160.
65. Madden B, Madden P, Powers S, Himsolt M. Portable graph layout and editing. *Graph Drawing (Proceedings of GD '95)* (*Lecture Notes in Computer Science*, vol. 1027), Brandenburg FJ (ed.). Springer-Verlag, 1996; 385–395.
66. Frick A. Upper bounds on the number of hidden nodes in Sugiyama's algorithm. *Graph Drawing (Proceedings of GD '96)* (*Lecture Notes in Computer Science*, vol. 1190), North S (ed.). Springer-Verlag, 1997; 169–183.
67. Eades P, McKay B, Wormald NC. On an edge crossing problem. *Proceedings of the 9th Australian Computer Science Conference*, 1986; 327–334.
68. Mäkinen E. Experiments on drawing 2-level hierarchical graphs. *Int. J. Comput. Math.* 1990; **36**(3+4):175–181.
69. Mutzel P. An alternative method to crossing minimization on hierarchical graphs. *Graph Drawing (Proceedings of GD '96)* (*Lecture Notes in Computer Science*, vol. 1190), North S (ed.). Springer-Verlag, 1997; 318–333.
70. Mutzel P, Weiskircher R. Two-layer planarization in graph drawing. *Algorithms and Computation (Proceedings of ISAAC'98)* (*Lecture Notes in Computer Science*, vol. 1533), Chwa K-Y, Ibarra OH (eds.). Springer-Verlag, 1998; 69–78.
71. Jünger M, Lee EK, Mutzel P, Odenthal T. A polyhedral approach to the multi-layer crossing minimization problem. *Graph Drawing (Proceedings of GD '97)* (*Lecture Notes in Computer Science*, vol. 1353), Di Battista G, (ed.) Springer-Verlag, 1997; 13–24.
72. Gansner ER, Koutsofios E, North SC, Vo KP. A technique for drawing directed graphs. *IEEE Trans. Softw. Eng.* 1993; **SE-19**(3):214–230.
73. Gansner ER, North S, Vo KP. DAG—A program that draws directed graphs. *Softw. Pract. Exp.* 1988; **18**(11):1047–1062.
74. Buti L, Di Battista G, Liotta G, Tassinari E, Vargiu F, Vismara L. GD-Workbench: A system for prototyping and testing graph drawing algorithms. *Graph Drawing (Proceedings of GD '95)* (*Lecture Notes in Computer Science*, vol. 1027), Brandenburg FJ (ed.). Springer-Verlag, 1996; 111–122.
75. Rosenstiehl P, Tarjan RE. Rectilinear planar layouts and bipolar orientations of planar graphs. *Discrete Comput. Geom.* 1986; **1**(4):343–353.

76. Tamassia R, Tollis IG. A unified approach to visibility representations of planar graphs. *Discrete Comput. Geom.* 1986; **1**(4):321–341.
77. Di Battista G, Tamassia R, Tollis IG. Constrained visibility representations of graphs. *Inform. Process. Lett.* 1992; **41**(1):1–7.
78. Garg A, Tamassia R. Upward planarity testing. *Order* 1995; **12**(2):109–133.
79. Rival I. Reading, drawing, and order. *Algebras and Orders*, Rosenberg IG, Sabidussi G (eds.). Kluwer Academic Publishers, 1993; 359–404.
80. Bertolazzi P, Cohen RF, Di Battista G, Tamassia R, Tollis IG. How to draw a series-parallel digraph. *Int. J. Comput. Geom. Appl.* 1994; **4**(4):385–402.
81. Chrobak M, Goodrich MT, Tamassia R. Convex drawings of graphs in two and three dimensions. *Proceedings of the 12th Annual ACM Symposium Comput. Geom.*, 1996; 319–328.
82. Garg A, Tamassia R. Planar drawings and angular resolution: Algorithms and bounds. *Algorithms (Proceedings of ESA'94)* (*Lecture Notes in Computer Science*, vol. 855), van Leeuwen J (ed.). Springer-Verlag, 1994; 12–23.
83. North S. 5114 directed graphs, 1995. ftp://ftp.research.att.com/dist/drawdag/dg.gz.
84. Himsolt M. GraphEd: A graphical platform for the implementation of graph algorithms. *Graph Drawing (Proceedings of GD '94)* (*Lecture Notes in Computer Science*, vol. 894), Tamassia R, Tollis IG (eds.). Springer-Verlag, 1995; 182–193.

