ELSEVIER

Computational Geometry 7 (1997) 303–325

Computational Geometry Theory and Applications

An experimental comparison of four graph drawing algorithms [☆]

Giuseppe Di Battista [a,*], Ashim Garg [b,1], Giuseppe Liotta [b,2], Roberto Tamassia [b,3], Emanuele Tassinari [c,4], Francesco Vargiu [c,5]

[a] *Dip. Informatica e Automazione, Università di Roma Tre, via della Vasca Navale 84, 00146 Roma, Italy*
[b] *Department of Computer Science, Brown University, 115 Waterman Street, Providence, RI 02912-1910, USA*
[c] *Dip. Informatica e Sistemistica, Università di Roma "La Sapienza", Via Salaria, 113 00198 Roma, Italy*

Communicated by E. Welzl; submitted 15 May 1995; revised 24 November 1995

Abstract

In this paper we present an extensive experimental study comparing four general-purpose graph drawing algorithms. The four algorithms take as input general graphs (with no restrictions whatsoever on connectivity, planarity, etc.) and construct orthogonal grid drawings, which are widely used in software and database visualization applications. The test data (available by anonymous ftp) are 11,582 graphs, ranging from 10 to 100 vertices, which have been generated from a core set of 112 graphs used in "real-life" software engineering and database applications. 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., crossings, bends, edge length) and running time. © 1997 Elsevier Science B.V.

Keywords: Graph drawing; Orthogonal drawings; Algorithms implementation; Experimental comparison of algorithms

[☆] Research supported in part by the US National Science Foundation under grant CCR-9423847, by the US Army Research Office under grant DAAH04-96-1-0013, by the "Progetto Ambienti di Supporto all Progettazione di sistemi Informativi" and grant 94.23.CT07 of the Italian National Research Council (CNR), by the NATO-CNR Advanced Fellowship Programme, and by the ESPRIT II Basic Research Actions Program of the European Community (project Algorithms and Complexity). An extended abstract of this work was presented at the 11th Annual ACM Symposium on Computational Geometry, held in Vancouver (CA) in June 1995.

[*] Corresponding author. E-mail: dibattista@iasi.rm.cnr.it.

[1] E-mail: ag@cs.brown.edu.

[2] E-mail: gl@cs.brown.edu.

[3] E-mail: rt@cs.brown.edu.

[4] E-mail: tassinar@dis.uniroma1.it.

[5] E-mail: vargiu@dis.uniroma1.it.

1. Introduction

Graph drawing algorithms construct geometric representations of abstract graphs and networks. Because of the direct applications of graph drawing to advanced graphic user interfaces and visualization systems, and thanks to the many theoretical challenges posed by the interplay of graph theory and geometry, an extensive literature on the subject [13,43] has grown in the last decade.

Various graphic standards have been proposed for the representation of graphs in the plane. Usually, vertices are represented by points or simple geometric figures (e.g., rectangles, circles), and each edge (u, v) is represented by a simple open Jordan curve joining the points associated with the vertices u and v. A drawing is *planar* if no two edges cross. A graph is planar if it admits a planar drawing. An *orthogonal* drawing maps each edge into a chain of horizontal and vertical segments (see Figs. 2–3). A *grid* drawing is embedded in a rectilinear grid such that the vertices and bends of the edges have integer coordinates. Orthogonal drawings are widely used for graph visualization in many applications, including database systems (Entity-Relationship diagrams), software engineering (Data-Flow diagrams), and circuit design (circuit schematics).

1.1. Previous experimental work in graph drawing

Many graph drawing algorithms have been implemented and used in practical applications. Most papers in this area show sample outputs, and some also provide limited experimental results on small (with fewer than 100 graphs) test suites (see, e.g., [11,17,19,24,27,29] and the experimental papers in [43]). However, in order to evaluate the practical performance of a graph drawing algorithm in visualization applications, it is essential to perform extensive experimentations with input graphs derived from the application domain.

The performance of four planar straight-line drawing algorithms [8,9,12,36,46] is compared in [23]. These algorithms have been implemented and tested in 10,000 randomly generated maximal planar graphs. The standard deviations in angle size, edge length, and face area are used to compare the quality of the planar straight-line drawings produced. Since the experiments are limited to randomly generated maximal planar graphs, this work gives only partial insight on the performance of the algorithms on general planar graphs.

Himsolt [20] presents a comparative study of twelve graph drawings algorithms, including [8,12,17,28,38,39,48,50]. The algorithms selected are based on various approaches (e.g., force-directed, layering, and planarization) and use a variety of graphic standards (e.g., orthogonal, straight-line, polyline). Only three algorithms draw general graphs, while the others are specialized for trees, planar graphs, Petri nets, and graph grammars. The experiments are conducted with the graph drawing system GraphEd [21]. Many examples of drawings constructed by the algorithms are shown, and various objective and subjective evaluations on the aesthetic quality of the drawings produced are given. However, statistics are provided only on the edge length, with few details on the experimental setting. The charts on the edge length have marked oscillations, due to the small size of the test suite (about 100 graphs). This work provides an excellent overview and comparison of the main features of some popular drawing algorithms. However, it does not give detailed statistical results on their performance.

After the conference version of the present paper appeared [4], Brandenburg and Rohrer [7] have compared five "force-directed" methods for constructing straight-line drawings of general undirected graphs. The algorithms are tested on a wide collection of examples and with different settings of the force parameters. The quality measures evaluated are crossings, edge length, vertex distribution, and running time. They also identify trade-offs between the running time and the aesthetic quality of the drawings produced.

Jünger and Mutzel [26] recently investigated crossing minimization strategies for straight-line drawings of 2-layer graphs, and compared the performance of eight popular heuristics for this problem.

1.2. Our results

In this paper we present an extensive experimental study comparing four general-purpose graph drawing algorithms. The four algorithms, denoted `Bend-Stretch`, `Column`, `Giotto` and `Pair`, are derived from theoretical papers [6,34,39,42], take as input general graphs (with no restrictions whatsoever on connectivity, planarity, etc.), and construct orthogonal grid drawings. The test data (available by anonymous ftp) are 11,582 graphs, ranging from 10 to 100 vertices, which have been generated from a core set of 112 graphs used in "real-life" software engineering and database applications. 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., crossings, bends, edge length) and running time.

The contributions of this work can be summarized as follows.

- We have developed a general experimental setting for comparing the practical performance of orthogonal drawing algorithms for general graphs.
- We have generated a large test suite of graphs derived from "real-life" software engineering and database applications. We believe that it will be useful to other researchers interested in experimental graph drawing.
- We have implemented four algorithms with solid theoretical foundations that construct orthogonal grid drawings of arbitrary input graphs.
- We have presented the first extensive experimental study of general-purpose graph drawing algorithms. The properties of the test suite, average values of the quality measures, and run times are summarized in 18 charts.
- We have found out that the observed average values of the area and number of bends are considerably lower than the worst-case bounds given by the theoretical analysis.

1.3. Organization of the paper

The rest of this paper is organized as follows. The four drawing algorithms analyzed are described in Section 2. Details on the experimental setting are given in Section 3. In Section 4, we summarize our experimental results by means of charts and perform a comparative analysis of the performance of the algorithms. Finally, open problems are addressed in Section 5.

2. The drawing algorithms under evaluation

The four drawing algorithms considered in this paper, denoted `Bend-Stretch`, `Column`, `Giotto` and `Pair`, take as input general graphs (with no restrictions whatsoever on connectivity, planarity, etc.) and construct orthogonal drawings.

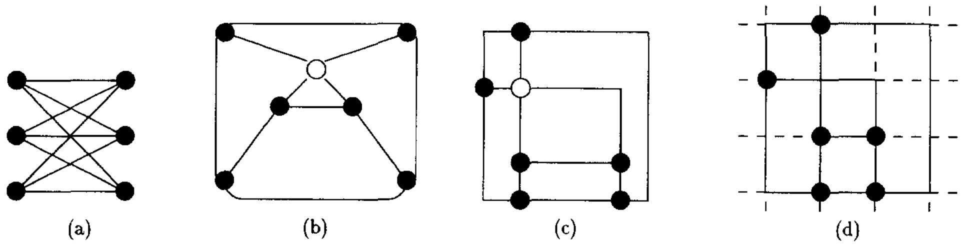


Fig. 1. A general strategy for orthogonal grid drawings. (a) Input graph. (b) Planarization. (c) Orthogonalization. (d) Compaction.

Algorithms `Bend-Stretch` and `Giotto` are based on a general approach where the drawing is incrementally specified in three phases (see Fig. 1). The first phase, *planarization*, determines the topology of the drawing. The second phase, *orthogonalization*, computes an orthogonal shape for the drawing. The third phase, *compaction*, produces the final drawing. This approach allows homogeneous treatment of a wide range of diagrammatic representations, aesthetics and constraints (see, e.g., [29,40,45]) and has been successfully used in industrial tools.

The main difference between the two algorithms is in the orthogonalization phase. Algorithm `Giotto` uses a network-flow method that guarantees the minimum number of bends but has quadratic time-complexity [39]. Algorithm `Bend-Stretch` adopts the "bend-stretching" heuristic [42] that only guarantees a constant number of bends on each edge but runs in linear time. More specifically, algorithms `Giotto` and `Bend-Stretch` are formally described in our graph drawing system (see Section 3.3) by the following *algorithmic paths* (sequences of steps and intermediate representations):

```
Giotto:

{multigraph
 {MakeConnected connected
  {MakePlanar connectedplanar
   {Make4planar fourplanar
    {Step3giotto orthogonal
     {Step4giotto manhattan}}}}}}
```

```
Bend-Stretch:

{multigraph
 {MakeConnected connected
  {MakeBiconnected biconnected
   {MakePlanar biconnectedplanar
    {Step1TaTo89 visibilityrepresentation
     {Step2TaTo89 orthogonal
      {Step4TaTo89 orthogonal
       {Step4Giotto manhattan}}}}}}}}
```

In the above notation, upper-case identifiers denote algorithmic components, and lower-case identifiers denote intermediate representations (such as graphs and drawings), as shown in Table 1.

Algorithm `Column` is an extension of the orthogonal drawing algorithm by Biedl and Kant [6] to graphs of arbitrary vertex degree. Algorithm `Pair` is an extension of the orthogonal drawing algorithm by Papakostas and Tollis [34,35] to graphs of arbitrary vertex degree. More specifically, algorithms `Column` and `Pair` are described by the following algorithmic paths:

```
Column:

{multigraph
 {MakeConnected connected
  {MakeBiconnected biconnected
   {BiedlKant manhattan}}}}
```

```
Pair:

{multigraph
 {MakeConnected connected
  {MakeBiconnected biconnected
   {PapakostasTollis manhattan}}}}
```

Table 1
Algorithmic components and intermediate representations used by algorithms `Bend-Stretch`, `Column`, `Giotto` and `Pair`

`multigraph`	general multigraphs accepted as input
`MakeConnected`	connectivity testing and augmentation
`connected`	connected graph
`MakeBiconnected`	biconnectivity testing and augmentation
`biconnected`	biconnected graph
`MakePlanar`	planarization [3,33]
`connectedplanar`	crossings are now replaced by dummy vertices
`Make4planar`	expansion of vertices with degree greater than 4 into rectangular symbols
`fourplanar`	viewing the rectangular symbols as cycles of dummy vertices, the graph has now maximum degree 4
`Step3giotto`	orthogonalization [39]
`step1TaTo89`	construction of a visibility representation [37,41,49]
`step2TaTo89`	fast orthogonalization [42]
`step4TaTo89`	bend-stretching transformations, which remove bends by local layout modifications [42]
`orthogonal`	orthogonal representation, describing shape of the drawing in term of its angles
`Step4giotto`	tidy compaction [3]
`BiedlKant`	orthogonal drawing algorithm [6]
`PapakostasTollis`	orthogonal drawing algorithm of [34,35]
`manhattan`	the final output is an orthogonal grid drawing

Note that the connectivity and biconnectivity augmentation steps were introduced in order to use algorithms designed for connected and biconnected graphs, respectively. The augmentation edges are not displayed in the final drawing.

Details on key algorithmic components of `Bend-Stretch`, `Column`, `Giotto` and `Pair` and their implementation are given below.

`MakePlanar`. This algorithmic component is the planarization phase of the algorithm described in [3]. It consists of the following steps.

Extraction of a Planar Subgraph. In this step, a set of edges is removed such that the resulting subgraph is planar. The technique used is a greedy heuristic [33] that extends the Hopcroft–Tarjan planarity testing algorithm [22] as follows: whenever a nonplanar configuration is detected, a sufficient number of edges is removed to resume the planarity testing algorithm.

Embedding of the Planar Subgraph. An embedding of the planar subgraph extracted by the previous step is constructed using a method that heuristically tries to minimize the nesting of the biconnected components.

Reinsertion of the Nonplanar Edges. Each "nonplanar edge" is reinserted by minimizing each time the number of edges crossed. This is done by solving a shortest path problem on the dual graph.

`Make4planar`. For each vertex v with degree $\deg(v) > 4$, expand vertex v into a *skeleton subgraph*, as follows: if $\deg(v) = 5$ or 6 then v is replaced by two new vertices connected by a new edge; if $\deg(v) > 6$, then v is replaced by a cycle with just enough vertices to accommodate the incident edges of v.

`Step3giotto`. The orthogonal representation is constructed using the constrained bend-minimization algorithm by Tamassia [39]. This algorithm computes a minimum cost flow on a network whose nodes represent the vertices and the faces of the graph, and whose edges represent the incidence relationships face-edge-vertex. The minimum cost flow is computed with the standard method of augmenting the flow along minimum-cost paths. Suitable constraints force a rectangular shape for the skeleton subgraphs (see `Make4planar`).

`Step4giotto`. This algorithmic component is the "tidy compaction" described in [3,40] that computes an orthogonal grid drawing from an orthogonal representation by assigning integer lengths to the horizontal and vertical segments of the edges. This is done in two steps. First, the faces of the orthogonal representation are decomposed into rectangles. Second, the lengths of the horizontal and vertical segments in the resulting rectangular floorplan are computed by means of a transformation into a pair of minimum cost flow problems. Each node of the flow network is associated with a rectangle in the drawing, and the flow conservation rule corresponds to the equality of the lengths of opposite sides. This step heuristically attempts at minimizing the area and the total edge length.

`MakeConnected`. A minimal set of fictitious edges joining the connected components is added to the graph.

`MakeBiconnected`. A set of fictitious edges joining one biconnected component to the all the other components is added to the graph to ensure biconnectivity. The size of this set is at most twice the optimal one.

`BiedlKant`. The orthogonal grid drawing is incrementally constructed by adding the vertices one at a time. Namely, at each step a vertex v is added plus the edges connecting v to previously added vertices. Some columns of the grid are "reserved" to draw the remaining incident edges of v. Concerning the position of v, since one row is used for each vertex, the y-coordinate is immediately given by the order of visit of v, and the x-coordinate is the one of the reserved column of the incident edge of v that minimizes the number of bends introduced by the new edges. The implementation closely follows the description of the algorithm in [6].

`PapakostasTollis`. This step is implemented using the description of the algorithm of [34] given in [35]. The algorithm as described in [35] requires each vertex of the input graph to have degree exactly 4. Consistent with this requirement, this step first introduces dummy edges so that each vertex has degree at least four. It then computes an st-numbering using a depth first search. Using this st-numbering, each vertex v is classified either as a 1–3 vertex, or a 2–2 vertex, or a 3–1 vertex: v is a 1–3 vertex if it has exactly 1 incoming and at least 3 outgoing edges; v is a 2–2 vertex if it has at least 2 incoming and at least 2 outgoing edges; and v is a 3–1 vertex if it has at least 3 incoming edges and exactly 1 outgoing edge. A drawing is then constructed using this classification, following the algorithm of [35]. In the drawing, each vertex is represented as a box whose height and width is equal to the number of rows and columns needed to make its incident edges "go in" or "come out" of it. Since the algorithm of [35] leaves some choice in placing the outgoing edges of a vertex v, we have devised two heuristics to improve the drawing.

- The columns of the incoming edges of v are used first for placing the outgoing edges. New columns are therefore created only if, after reusing all columns of the incoming edges, there are still some outgoing edges left that need to be routed. This approach is a simple extension of the approach of [35], and is useful for placing vertices with degree more than 4.

- To reduce crossings, whenever possible, all the outgoing edges of v that require new columns are assigned to consecutive (new) columns next to the column where v is placed.

The rows and columns are maintained using balanced binary trees that allow efficient (logarithmic time) insertions.

Finally, after the drawing is constructed, a compaction step is carried out for reducing the area of the drawing. All the dummy edges introduced in the input graph are deleted. Each row or column of the drawing to which no edge of the input graph is assigned is also deleted.

Examples of "typical" drawings generated by `Bend-Stretch`, `Column`, `Giotto` and `Pair` are shown in Figs. 2–3. The drawings are all orthogonal grid drawings.

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 drawing constructed. 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`. Our implementation uses 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 (`Step{1,2,4}TaTo89`) 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, so that all the four algorithms heuristically attempt at reducing the number of crossings. However, algorithm `Giotto` guarantees to construct a planar drawing if the input graph is planar.

3. Experimental setting

3.1. *Quality measures analyzed*

The following quality measures of a drawing of a graph have been considered.

`Area`: area of the smallest rectangle with horizontal and vertical sides covering the drawing;
`Cross`: total number of crossings;
`TotalBends`: total number of bends;
`TotalEdgeLen`: total edge length;
`MaxEdgeBends`: maximum number of bends on any edge;
`MaxEdgeLen`: maximum length of any edge;
`UnifBends`: standard deviation of the number of bends on the edges;
`UnifLen`: 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.

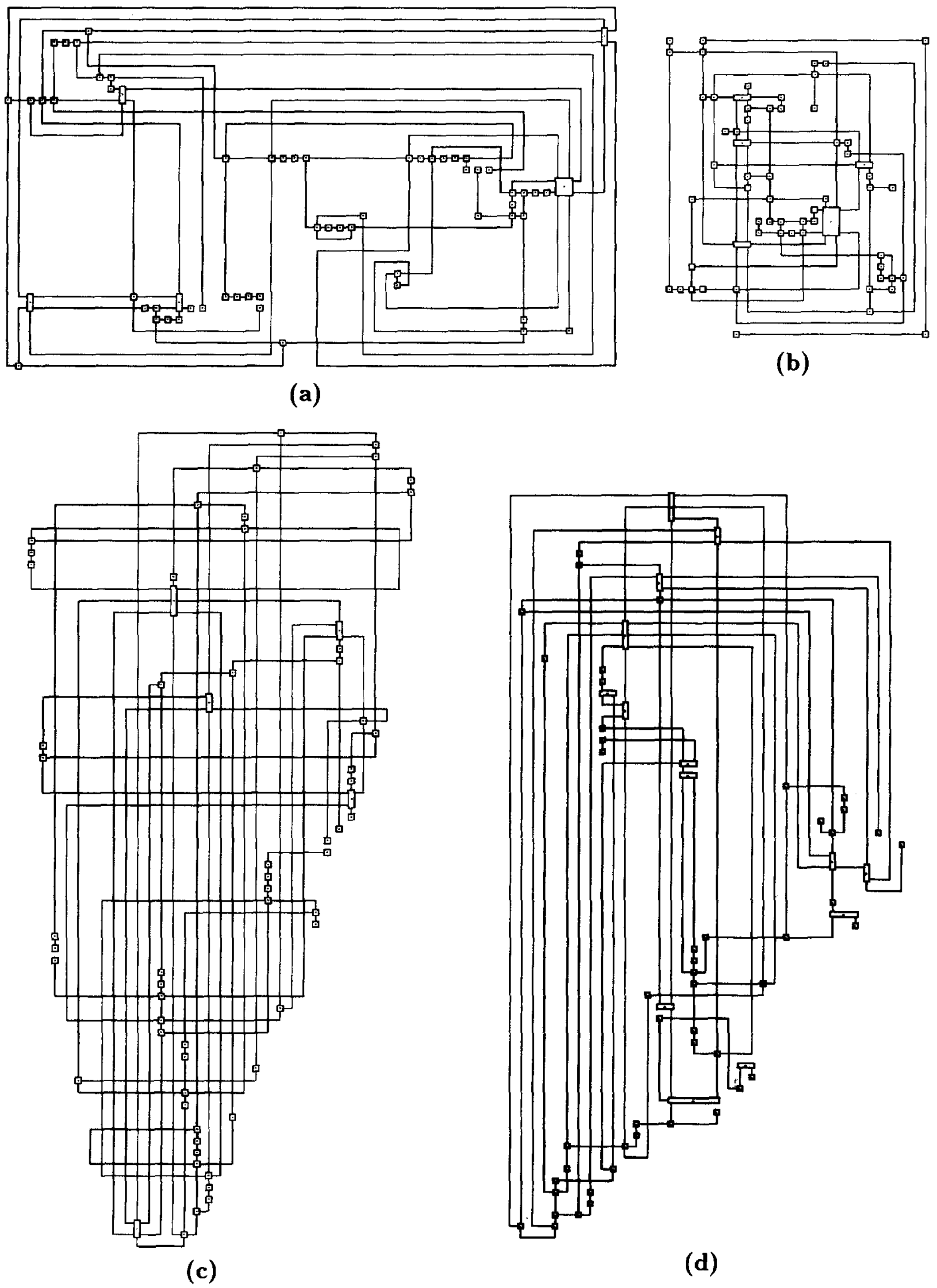

(a)
(b)
(c)
(d)

Fig. 2. Drawings of the same 63-vertex graph produced by algorithms (a) Bend-Stretch, (b) Giotto, (c) Column and (d) Pair, respectively.

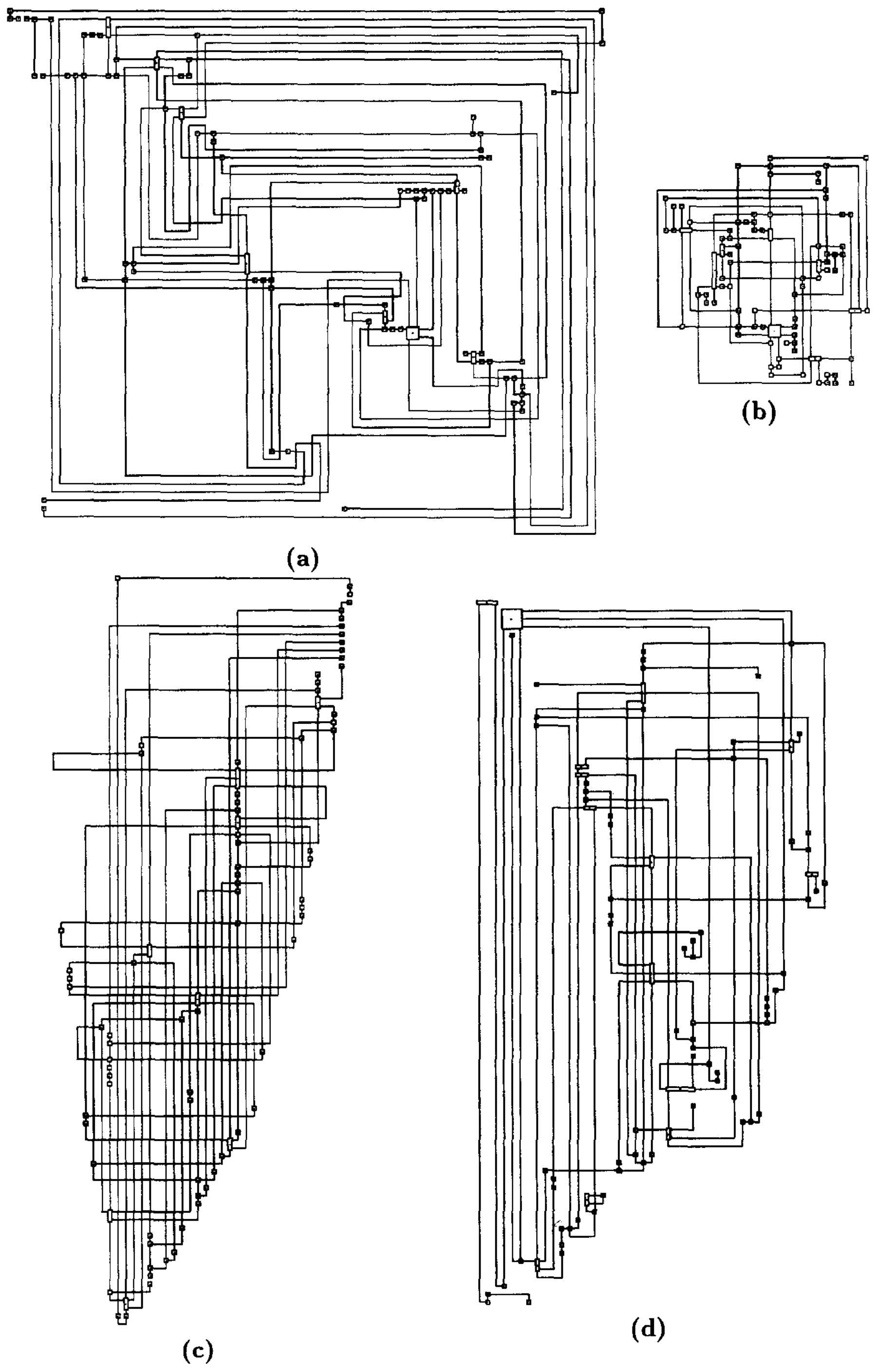


Fig. 3. Drawings of the same 85-vertex graph produced by algorithms (a) Bend-Stretch, (b) Giotto, (c) Column and (d) Pair, respectively.

It is widely accepted (see, e.g., [13]) that small values of the above measures are related to the perceived aesthetic appeal and visual effectiveness of the drawing.

3.2. Generation of the test graphs

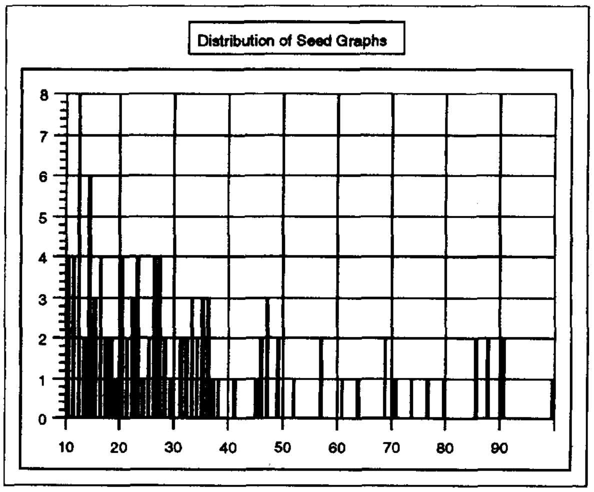
Since we are interested in evaluating the performance of graph drawing algorithms in practical applications, we have disregarded approaches completely based on random graphs.

Our test graph generation strategy is as follows. First, we have focused on the important application area of database and software visualization, where Entity-Relationship diagrams and Data-Flow diagrams are usually displayed with orthogonal drawings.

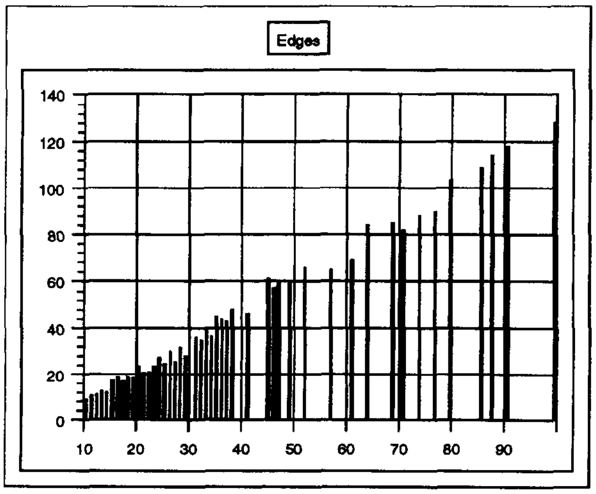
Second, we have collected 112 "real life" graphs with number of vertices between 10 and 100, from now on called *core graphs*, from the following sources:

- 54% of the graphs have been obtained from major Italian software companies (especially from *Database Informatica*) and large government organization (including the Italian Internal Revenue Service and the Italian National Advisory Council for Computer Applications in the Government (*Autorità per l'Informatica nella Pubblica Amministrazione*));
- 33% of the graphs were taken from well-known reference books in software engineering [18] and database design [1], and from journal articles on software visualization in the recent issues of *Information Systems* and the *IEEE Transactions on Software Engineering*;
- 13% of the graphs were extracted from theses in software and database visualization written by students at the University of Rome "La Sapienza".

Fig. 4 depicts the distribution of the core graphs with respect to the number of vertices, and the average number of edges of the test graphs with N vertices, for $N = 10, \ldots, 100$.



(a)



(b)

Fig. 4. (a) Distribution of the core graphs with respects to the number of vertices. (b) Average number of edges of the core graphs versus number of vertices.

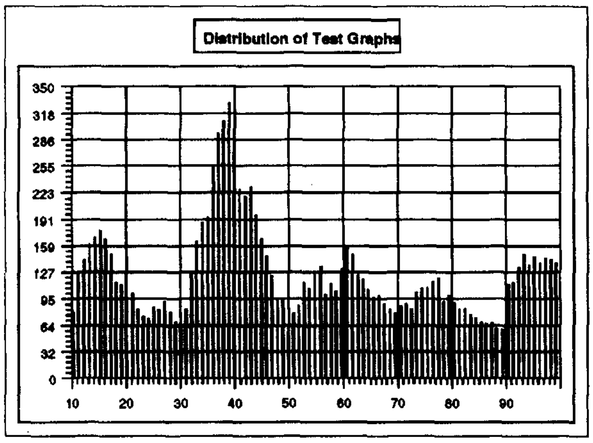
Third, we have generated the 11,582 test graphs as variations of the core graphs. This step is the most critical, since we needed to devise a method for generating graphs "similar" to the core graphs.

Our approach is based on the following scheme. We defined several primitive operations for updating graphs, which correspond to the typical operations performed by designers of Entity-Relationship and Data-Flow Diagrams, and attributed a certain probability to each of them. More specifically, the updating primitives we have used are the following: *InsertEdge*, which inserts a new edge between two existing vertices; *DeleteEdge*, which deletes an existing edge; *InsertVertex*, which splits an existing edge into two edges by inserting a new vertex; *DeleteVertex*, which deletes a vertex and all its incident edges; and *MakeVertex*, which creates a new vertex and connects it to a subset of vertices.

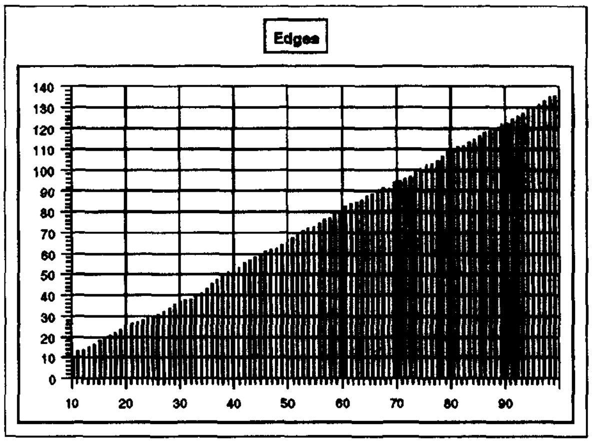
The test graphs were then generated in several iterations starting from the core graphs by applying random sequences of operations with a "genetic" mechanism. Namely, at each iteration a new set of test graphs was obtained by applying a random sequence of operations to the current test set. Each new graph was then evaluated for "suitability", and those found not suitable were discarded. The probability of each primitive operation was varied at the end of each iteration.

The evaluation of the suitability of the generated graphs was conducted using both objective and subjective analyses. The objective analysis consisted of determining whether the new graph had similar structural properties with respect to the core graph it was derived from. We have taken into account parameters like the average ratio between number of vertices and number of edges and the average number of biconnected components. The subjective analysis consisted in a visual inspection of the new graph and an assessment by expert users of Entity-Relationship and Data-Flow diagrams of its similarity to a "real-life" diagram. For obvious reasons, the subjective analysis has been done on a randomly selected subset of the graphs.

Fig. 5 depicts the distribution of the test graphs with respect to the number of vertices, and the average number of edges of the test graphs with N vertices, for $N = 10, \ldots, 100$. At least 50 test graph for each vertex cardinality between 10 and 100 have been generated. The average number of



(a)



(b)

Fig. 5. (a) Distribution of the test graphs with respect to the number of vertices. (b) Average number of edges of the test graphs versus number of vertices.

edges of the test graphs is slightly higher than the one of the core graphs. The 11,582 test graphs are available on the Internet from `ftp://infokit.dis.uniroma1.it/public/`.

Sparsity and "near-planarity" are typical properties of graphs used in software engineering and database applications [2]. As expected, the test graphs turn out to be sparse (the average vertex degree is about 2.7, see Fig. 5(b)) and with low crossing number (the experiments show that the average number of crossings is no more than about 0.7 times the number of vertices, see Fig. 6(b)). We did not include graphs with more than 100 vertices because they are rarely displayed in full in the above applications (clustering methods are typically used to hierarchically display large graphs).

3.3. Diagram server

Our experimental study was conducted using *Diagram Server* [14,15], a network server for client-applications that use diagrams (drawings of graphs). Diagram Server offers to its clients a set of facilities to represent and manage diagrams through a multiwindowing environment. One of the most important facilities is a library of automatic graph drawing algorithms [5]. A graph drawing algorithm is fully specified in the automatic graph drawing facility by an *algorithmic path*, which describes the sequence of steps and intermediate representations (e.g., planar embedding, orthogonal shape, visibility representation) produced by the algorithm. Diagram Server can be customized according to different application contexts and graphic environments.

Diagram Server has been implemented and tested on a wide variety of client-applications including information systems design, project management and reverse software engineering. Diagram Server is a noncommercial academic prototype developed at the University of Rome.

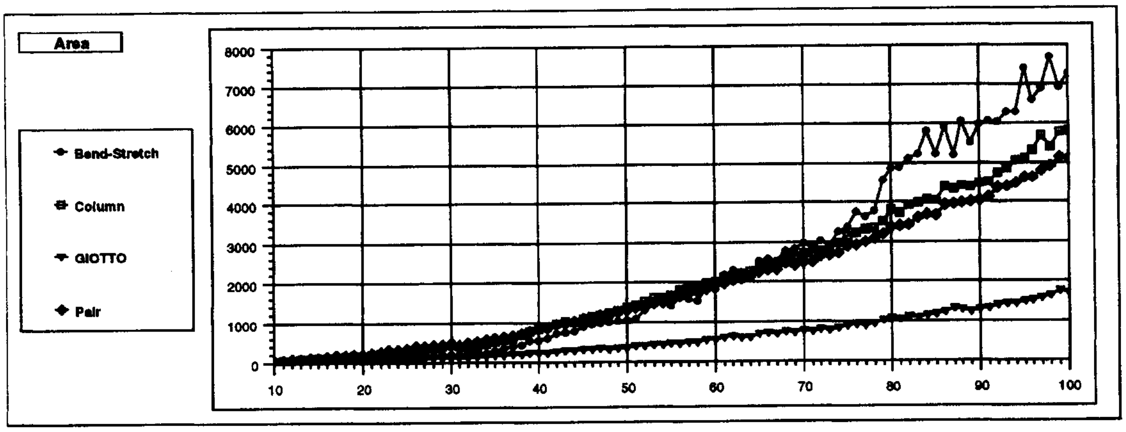
4. Analysis of the experimental results

The `Bend-Stretch`, `Column`, `Giotto` and `Pair` algorithms have been executed on each of the 11,582 test graphs. Figs. 6–8 show the average values of the nine quality measures `Area`, `Cross`, `ScreenRatio`, `TotalBends`, `MaxEdgeBends`, `UnifBends`, `TotalEdgeLen`, `MaxEdgeLen`, `UnifLen` achieved by the four algorithms on test graphs with N vertices, for $N = 10, \ldots, 100$.

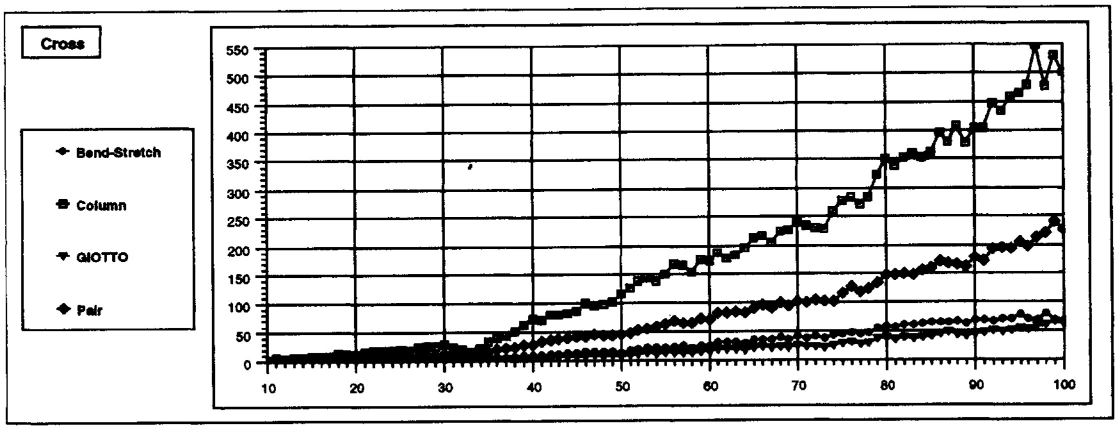
While for some of the quality measures, like `Area`, `Time` and `TotalBends`, it is possible to compare the experimental results with the theoretical performance of the algorithms, for some other quality measures the literature does not provide an explicit analysis. In what follows we describe the experimental results and provide a comparative analysis of them.

`Area`. `Giotto` outperforms all three of `Bend-Stretch`, `Column` and `Pair`. For $N > 70$, the difference becomes significant (see, e.g., Fig. 3). `Bend-Stretch`, `Column` and `Pair` behave the same for $N < 75$. Consistently with the theoretical analysis, `Pair` performs either the same or better than `Column` for each N. For $N > 75$, `Bend-Stretch` performs better than `Pair` and `Column`. This result is somehow surprising since one would expect a better behavior for `Pair` and `Column`, which allow edge crossings even when the input graph is planar (the area of a drawing of a planar graph benefits of the possibility of introducing edge-crossings [32,47]). We observe the following.

(a)



(b)



(c)

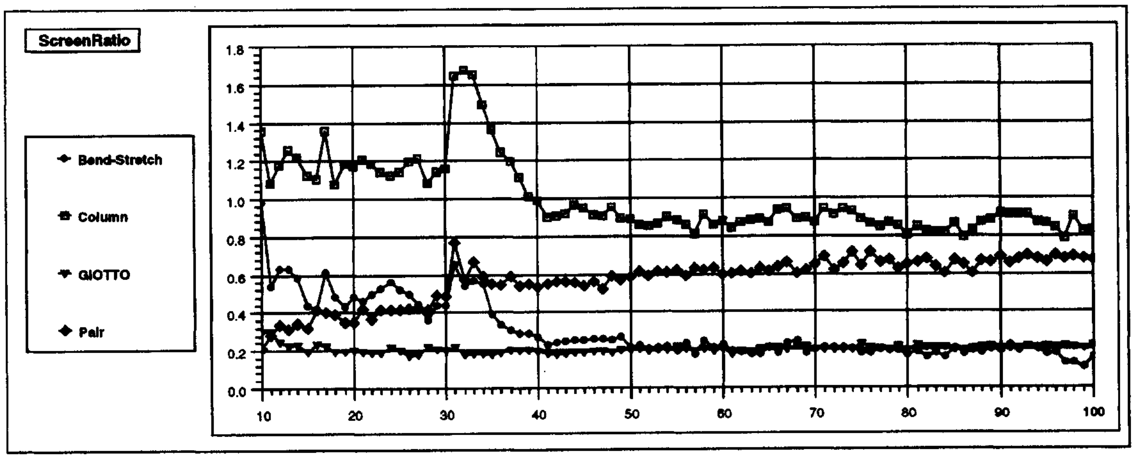
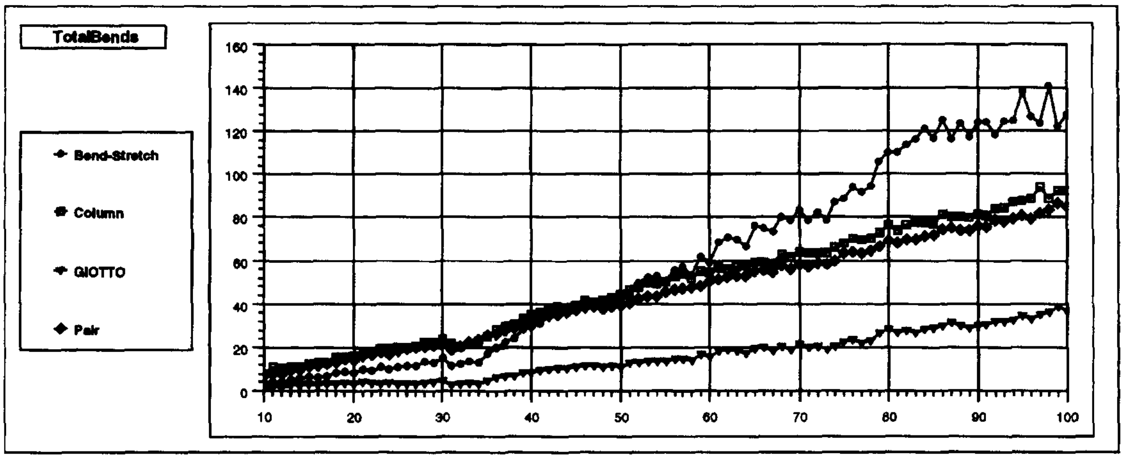
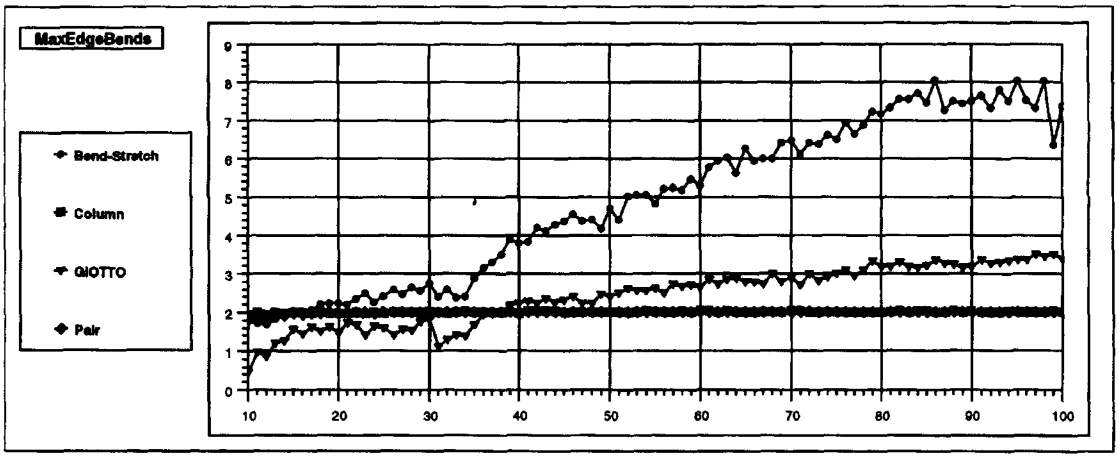


Fig. 6. (a) Average area versus number of vertices. (b) Average number of crossings versus number of vertices. (c) Average deviation from the screen ratio versus number of vertices.

(a)



(b)



(c)

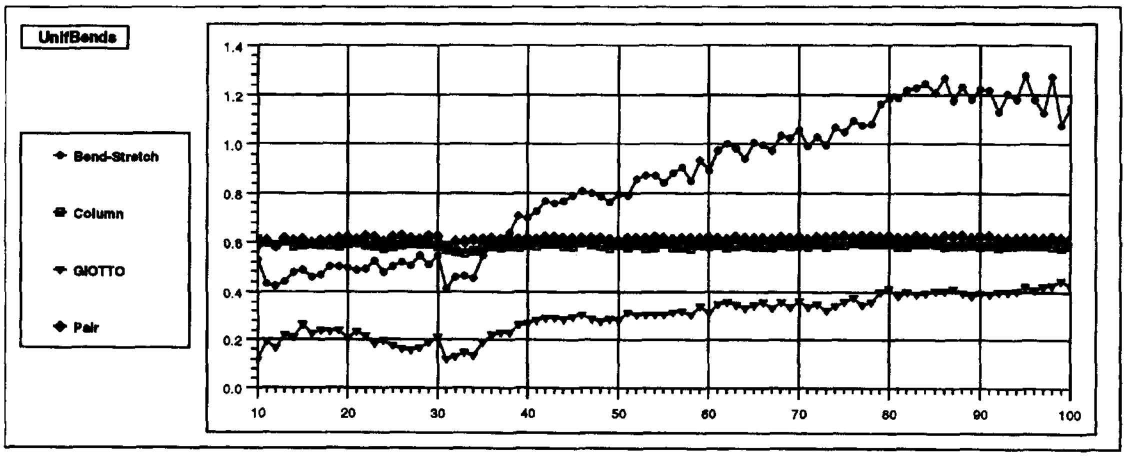
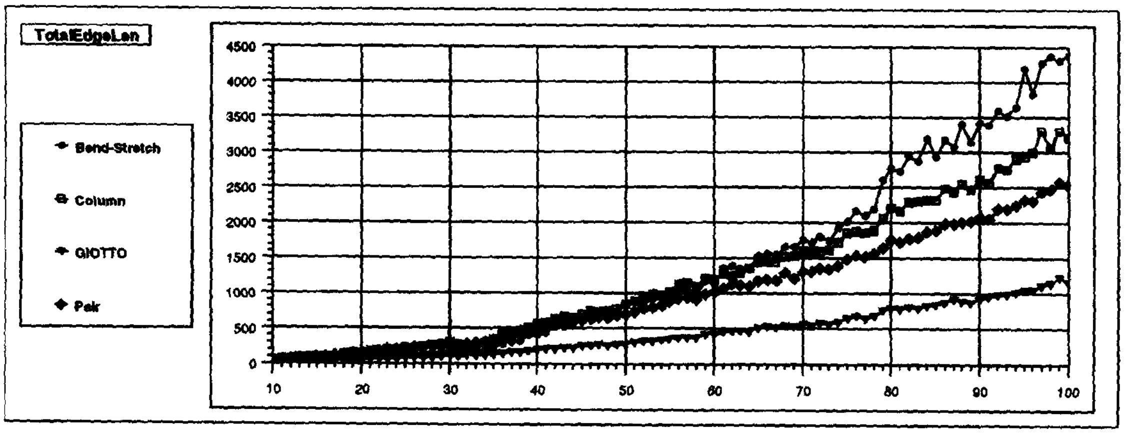
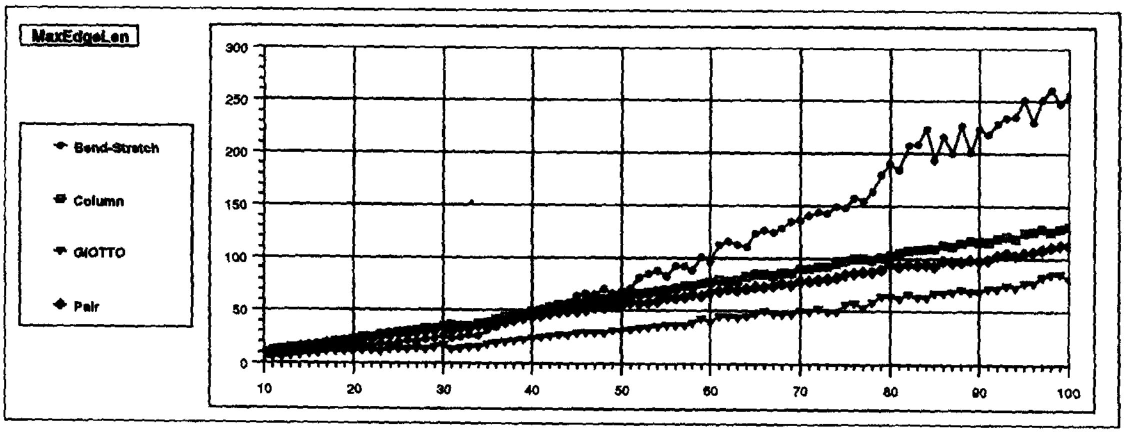


Fig. 7. (a) Average total number of bends versus number of vertices. (b) Average maximum number of bends on any edge versus number of vertices. (c) Average standard deviation of the number of bends on the edges versus number of vertices.

(a)



(b)



(c)

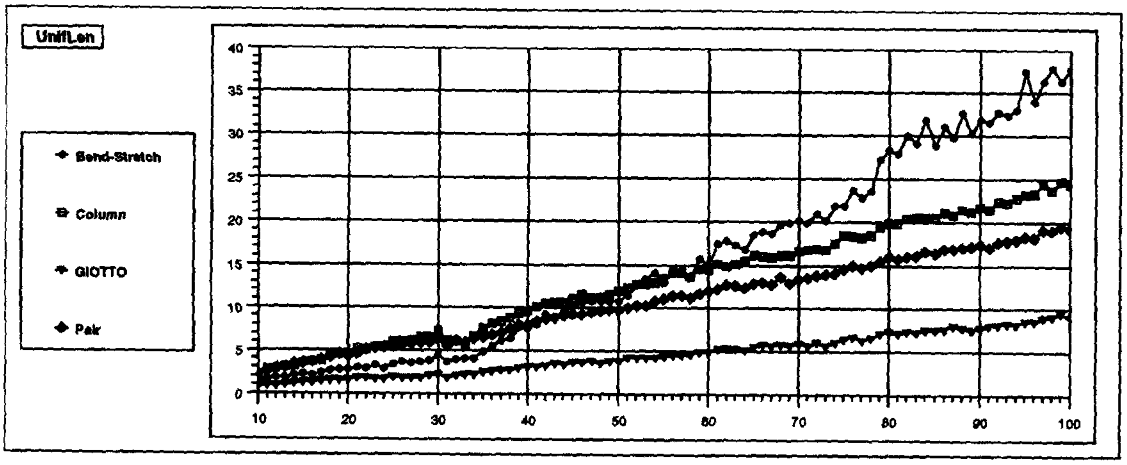


Fig. 8. (a) Average total edge length versus number of vertices. (b) Average maximum edge length versus number of vertices. (c) Average standard deviation of the edge length versus number of vertices.

Observation 1. 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`.

Observation 2. `Bend-Stretch`, `Pair` and `Column` need to transform the input graph into a directed acyclic graph by means of an *st*-numbering procedure. The *st*-numbering method implemented in Diagram Server is based on a depth-first search of the input graph. An alternative approach could be to use a breadth-first search. A very recent experimental study [44] shows that the choice of an *st*-numbering based on depth-first search can have a negative effect on the performance of `Bend-Stretch`, `Pair` and `Column` with respect to several quality measures, including `Area`, `Cross`, `ScreenRatio` and `TotalEdgeLen`. In fact the *st*-numbering based on the depth-first search tends to create edges that connect vertices whose positions in the drawing are far apart vertically. Thus, the drawing contains long edges that may occupy several grid cells and may cross several other edges. Conversely, if the *st*-numbering is based on breadth-first search, the edges tend to be shorter, and tend to have their endpoints on consecutive horizontal layers in the drawing.

Observation 3. `Bend-Stretch`, `Pair` and `Column` transform the input graph into a biconnected graph by means of the method `MakeBiconnected`. The edges (dummy edges) that are introduced by this method are deleted only in the final drawing. In most cases, deleting dummy edges in a drawing does not reduce its overall area. Because biconnectivity is needed only for computing the *st*-numbering, we 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. We believe that such an approach can also improve quality measures `Cross` and `TotalBends`.

Observation 4. `Bend-Stretch` and `Giotto` are based on algorithms designed for planar graphs, so their behavior strongly depends on the planarization step of the corresponding algorithmic path (`Makeplanar` step). The planarization step replaces crossings with dummy vertices. If the number of such dummy vertices is not too large, then the total number of vertices (original vertices plus dummy vertices) in the planarized graph does not differ too much from the number of vertices of the input graph. Although in the worst case the number of crossings (i.e., of dummy vertices) can be quadratic in the number of vertices of the input graph, the `Makeplanar` step has a very good behavior in most cases. Namely, observing the curves relative to the quality measure `Cross`, it is easy to see that the number of dummy vertices for `Giotto` and `Bend-Stretch` is always less than the number of vertices of the input graph. On the other hand, `Column` and `Pair` do not try to minimize the number of crossings along the edges. Since each crossing occupies a unit cell on the grid, the area of the drawings produced by `Giotto` and `Bend-Stretch` is positively affected by the good performance of the `Makeplanar` step.

`Cross`. `Bend-Stretch` and `Giotto` behave more or less the same, especially for $10 < N < 60$, and $95 < N < 100$. `Bend-Stretch` is in general slightly worse than `Giotto` because of its `MakeBiconnected` step that makes the input graph denser before the `Makeplanar` step is applied. The very different behavior of `Pair` and `Column` can be explained with Observations 2, 3 and 4.

ScreenRatio. The behavior of Giotto is very good in the whole interval. The behavior of Bend-Stretch is about the same as that of Giotto between 40 and 100. The behavior of Column is unsatisfactory. The screen ratio of the drawings produced by all 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 curves 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 (see, e.g., Figs. 2–3). As pointed out in Observation 2, an st-numbering based on a breadth-first search could improve the performances of Bend-Stretch, Column and Pair, because it would avoid long edges that stretch the drawing in one dimension.

TotalBends. The experimental results sharply fit the theoretical results. Namely, Giotto has the minimum number of bends; Bend-Stretch, Column and Pair have a number of bends that, also for the constants, is essentially the one predicted by the theoretical analysis [6,34,35,42]. As pointed out in Observation 1, the performance of Bend-Stretch, Pair and Column on TotalBends negatively affects their performance on Area. Furthermore, we believe that TotalBends is penalized by the dummy edges introduced by the Makebiconnected step (see Observation 3), which force several edges in the drawing to have apparently unnecessary bends (see, e.g., Figs. 2–3).

MaxEdgeBends. 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, we would have expected 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 MakePlanar step into several fragments, and the bound of two bends per edge applies separately to each fragment. Finally, we note that the average value of MaxEdgeBends is always less than 4 for Giotto, which suggests that the theoretical $O(N)$ worst-case value of MaxEdgeBends is rarely attained. This suggests studying theoretical bounds.

UnifBends. 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 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.

TotalEdgeLen. 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 notice the similarity of the curves for TotalEdgeLen and Area, which fits with the intuitive notion that small TotalEdgeLen and Area generally go together. All the observations made for Area (Observations 1, 2, 3 and 4) can be applied to this case.

MaxEdgeLen. 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 $N > 45$. We believe the length of the longest edge can be reduced for Bend-Stretch for Pair and for Column by implementing their *st*-numbering procedure based on a breadth first search (Observation 2).

UnifLen. 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 $N > 60$. Notice that the curves for UnifLen and those for TotalEdgeLen are very similar. This reflects the fact that numerically UnifLen is affected by the value of TotalEdgeLen.

Time. The experiments have been performed on Sun Sparc-10 workstations. The implementation of the algorithms uses methods that are efficient in practice but are not asymptotically optimal. Our implementation of both Column and Pair is quite fast, which is consistent with their theoretical low 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 our opinion, this is so because for small graphs ($N < 45$) the quadratic time complexity of step Step3giotto of Giotto dominates the linear time complexity of steps Step1TaTo89, Step2TaTo89 and Step4TaTo89 of Bend-Stretch, whereas for larger graphs ($N > 45$), Bend-Stretch is slower than Giotto for the following two reasons.

- Bend-Stretch makes the graph biconnected with the Makebiconnected step by introducing dummy edges. Hence, the Makeplanar step works on larger graphs for Bend-Stretch, than is does for Giotto.
- Since the total number of bends introduced by Bend-Stretch is larger than the one introduced by Giotto, step Step4giotto, which treats bends as dummy vertices, requires more time for Bend-Stretch than it does for Giotto.

The curves 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 is considerably slower than Column and Pair.

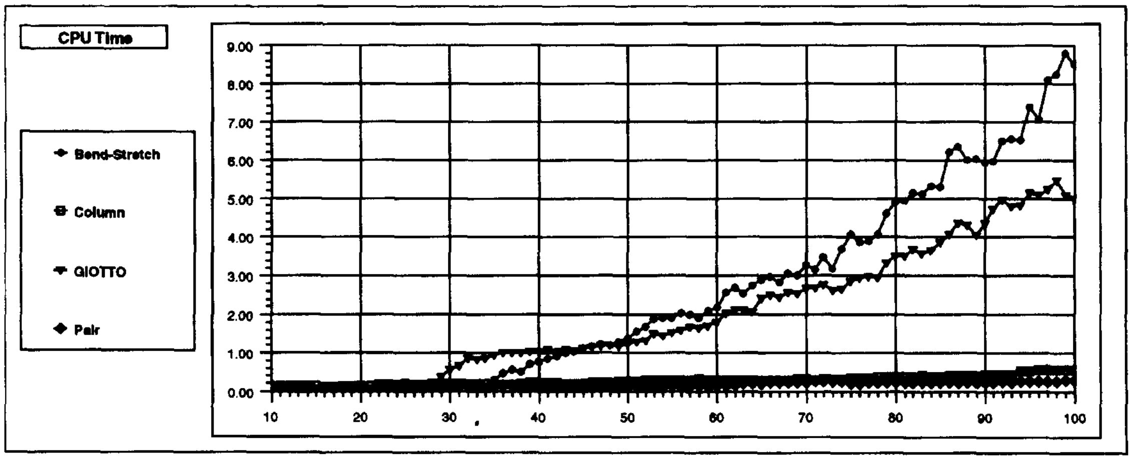
We have also performed the experiments on the 112 core graphs only. For example, Fig. 10 shows the average values of the three quality measures, Area, Cross and TotalEdgeLen, achieved by the four algorithms on the core graphs.

Clearly, due to the small number of core graphs, the charts in Fig. 10 have marked oscillations. Hence, one cannot derive general conclusions from them. The performance of the four algorithms on the core graphs is similar to that on the test graphs, except for algorithm Bend-Stretch, which does better on the core graphs than on the test graphs. We believe that this is due to the fact that the performance of algorithm Bend-Stretch is heavily dependent on the number of crossings computed in the planarization phase, which is smaller for the core graphs than for the test graphs (see Figs. 6(b) and 10(b)). Also, the average values of the quality measures for the core graphs are smaller than for the test graphs because the core graphs have slightly fewer edges than the test graphs (see Figs. 4(b) and 5(b)).

5. Conclusions and open problems

The main conclusion of our experimental study on four orthogonal grid drawing algorithms is that for a representative test suite of graphs derived from software engineering and database applications,

(a)



(b)

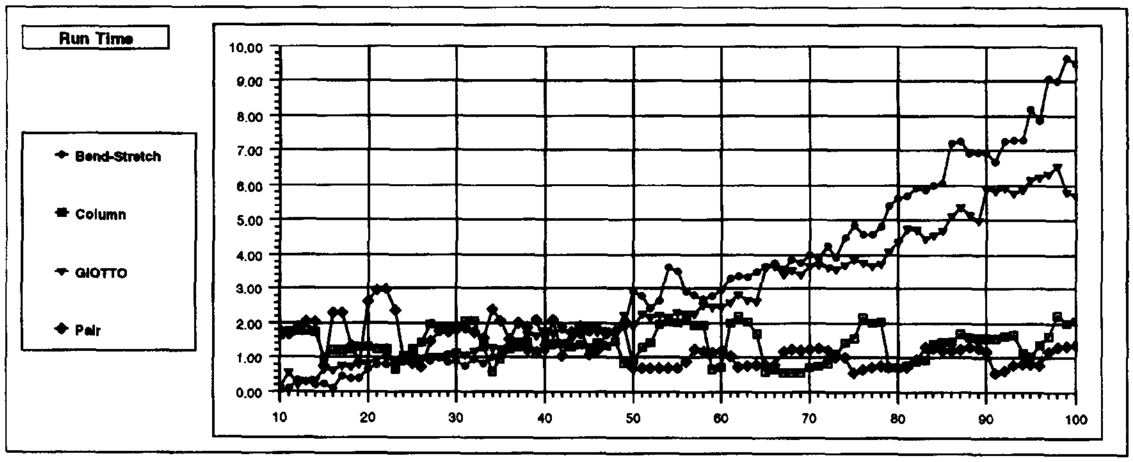


Fig. 9. (a) Average CPU time (seconds) versus number of vertices. (b) Average total run time (seconds) versus number of vertices.

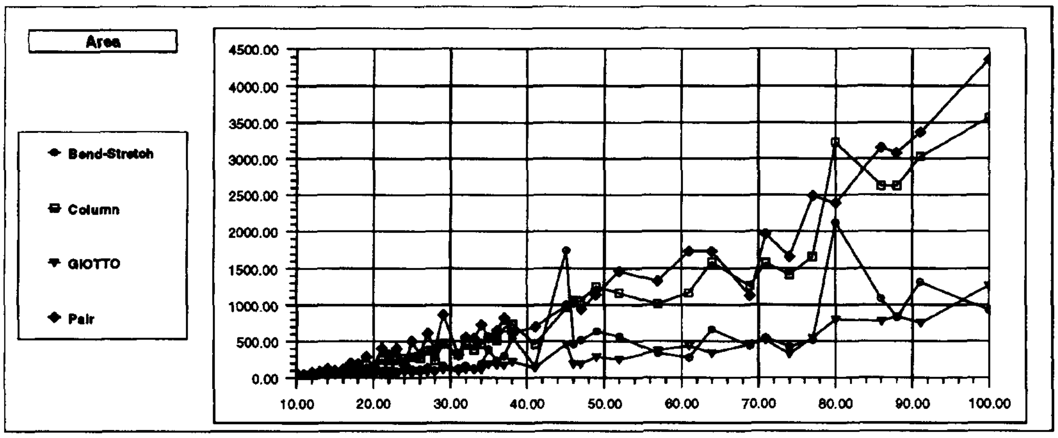
algorithm Giotto, which is based on a preliminary planarization step followed by an exact bend minimization step, outperforms for most quality measures the other algorithms, which either do not perform a preliminary planarization step (Column and Pair) or use a heuristic bend minimization method (Bend-Stretch). 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 in terms of a substantially higher running time.

Our work suggests the following considerations.

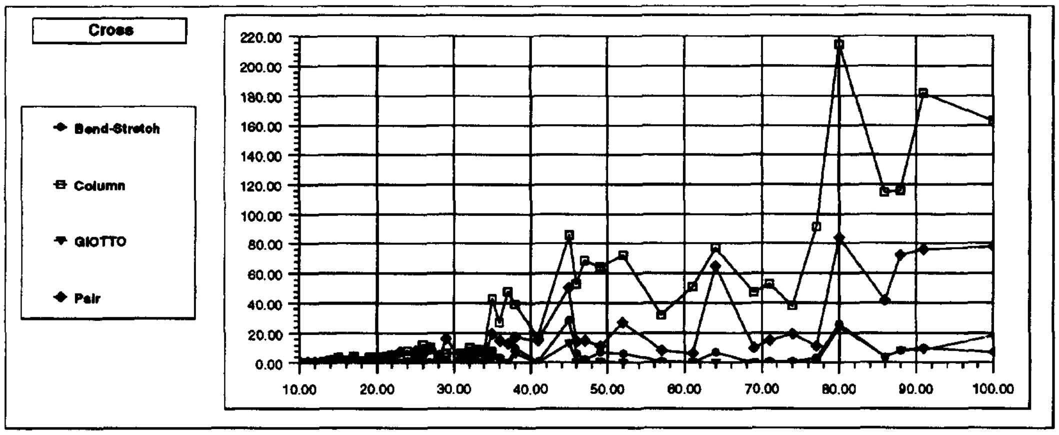
- Graph drawing has a tradition of combining theoretical and applied work. We believe that experimental studies are essential to strengthen this link.
- The theoretical analysis of drawing algorithms for special classes of graphs (e.g., degree-4, biconnected, planar) is insufficient to predict the behavior of algorithm for general graphs derived from them.

Finally, the experiments performed are an interesting source of both theoretical and practical open problems:

(a)



(b)



(c)

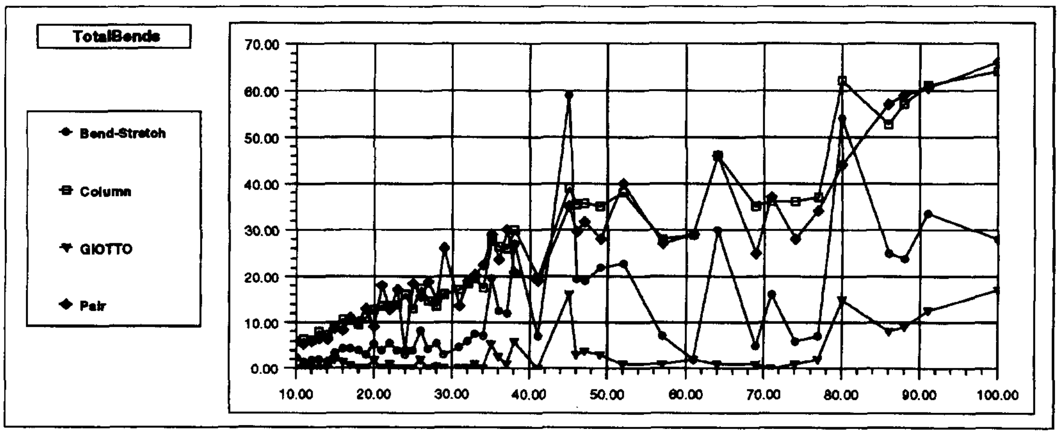


Fig. 10. (a) Average area versus number of vertices. (b) Average number of crossings versus number of vertices. (c) Average total number of bends versus number of vertices.

- It would be interesting to perform further experiments on the practical performance of separator-based methods [16,32,47] that were originally developed for VLSI layout.
- The behavior of `Bend-Stretch` could be improved by using, instead of the classical algorithms of [37,41,49], the algorithm by Kant [30] for constructing compact visibility representations.
- The performance of `Giotto` and `Bend-Stretch` is affected by the number of crossings introduced by the planarization phase. Can a more sophisticated heuristics (for example, based on the work of Jünger and Mutzel [24] on the computation of the maximum planar subgraph) dramatically improve the behavior of such algorithms?
- The performance of the algorithms `Bend-Stretch`, `Column` and `Pair` is affected by the biconnectivity augmentation step (`MakeBiconnected`). How much will it improve if we use a more sophisticated biconnectivity augmentation technique that preserves planarity (e.g., [31])? This issue is addressed in [25].
- One of the computational bottlenecks of the `Giotto` algorithm is the bend minimization step (`Step3giotto`), which has quadratic time complexity [39]. It would be interesting to improve on the time complexity of bend minimization.
- It would be interesting to devise practical algorithms for orthogonal drawings in the 3-dimensional space. For recent results, see [10].
- Extensive experiments on algorithms for constructing other types of drawings (e.g., straightline, polyline, upward) should be conducted.

Acknowledgements

We thank Yannis Tollis, Achilleas Papakostas and the anonymous referees for useful comments and suggestions.

References

[1] C. Batini, S. Ceri and S.B. Navathe, Conceptual Database Design, an Entity-Relationship Approach (Benjamin/Cummings, Menlo Park, CA, 1992).

[2] C. Batini, L. Furlani and E. Nardelli, What is a good diagram? A pragmatic approach, in: Proc. 4th Internat. Conf. on the Entity Relationship Approach (1985).

[3] C. Batini, E. Nardelli and R. Tamassia, A layout algorithm for data-flow diagrams, IEEE Trans. Software Engrg. 12 (4) (1986) 538–546.

[4] G. Di Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari and F. Vargiu, An experimental comparison of three graph drawing algorithms, in: Proc. 11th Ann. ACM Sympos. Comput. Geom. (1995) 306–315.

[5] P. Bertolazzi, G. Di Battista and G. Liotta, Parametric graph drawing, IEEE Trans. Software Engrg. 21 (8) (1995) 662–673.

[6] T. Biedl and G. Kant, A better heuristic for orthogonal graph drawings, in: Proc. 2nd Annu. European Sympos. Algorithms (ESA '94), Lecture Notes in Computer Science 855 (Springer, Berlin, 1994) 24–35.

[7] F.J. Brandenburg, M. Himsolt and C. Rohrer, An experimental comparison of force-directed and randomized graph drawing algorithms, in: Proc. Graph Drawing 1995, Lecture Notes in Computer Science 1027 (1996) 76–87.

[8] N. Chiba, K. Onoguchi and T. Nishizeki, Drawing planar graphs nicely, Acta Inform. 22 (1985) 187–201.

[9] M. Chrobak and T. Payne, A linear-time algorithm for drawing planar graphs, Inform. Process. Lett. 54 (1995) 241–246.

[10] R.F. Cohen, P. Eades, T. Lin and F. Ruskey, Three-dimensional graph drawing, in: R. Tamassia and I.G. Tollis, eds., Graph Drawing (Proc. GD '94), Lecture Notes in Computer Science 894 (Springer, Berlin, 1995) 1–11.

[11] R. Davidson and D. Harel, Drawing graphs nicely using simulated annealing, Technical Report, Department of Applied Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot (1989).

[12] H. de Fraysseix, J. Pach and R. Pollack, Small sets supporting Fary embeddings of planar graphs, in: Proc. 20th Ann. ACM Sympos. Theory Comput. (1988) 426–433.

[13] G. Di Battista, P. Eades, R. Tamassia and I.G. Tollis, Algorithms for drawing graphs: an annotated bibliography, Computational Geometry 4 (5) (1994) 235–282.

[14] G. Di Battista, A. Giammarco, G. Santucci and R. Tamassia, The architecture of Diagram Server, in: Proc. IEEE Workshop on Visual Languages (VL '90) (1990) 60–65.

[15] G. Di Battista, G. Liotta and F. Vargiu, Diagram Server, J. Visual Languages and Computing 6 (3) (1995) (Special issue on Graph Visualization, I.F. Cruz and P. Eades, eds.).

[16] D. Dolev, F.T. Leighton and H. Trickey, Planar embedding of planar graphs, in: F.P. Preparata, ed., Advances in Computing Research, Vol. 2 (JAI Press, Greenwich, CT, 1985) 147–161.

[17] T. Fruchterman and E. Reingold, Graph drawing by force-directed placement, Softw.—Pract. Exp. 21 (11) (1991) 1129–1164.

[18] G. Gane and T. Sarson, Structured Systems Analysis (Prentice-Hall, Englewood Cliffs, NJ, 1979).

[19] E.R. Gansner, S.C. North and K.P. Vo, DAG—a program that draws directed graphs, Softw.—Pract. Exp. 18 (11) (1988) 1047–1062.

[20] M. Himsolt, Comparing and evaluating layout algorithms within GraphEd, J. Visual Languages and Computing 6 (3) (1995) (Special issue on Graph Visualization, I.F. Cruz and P. Eades, eds.).

[21] M. Himsolt, GraphEd: a graphical platform for the implementation of graph algorithms, in: R. Tamassia and I.G. Tollis, eds., Graph Drawing (Proc. GD '94), Lecture Notes in Computer Science 894 (Springer, Berlin, 1995) 182–193.

[22] J. Hopcroft and R.E. Tarjan, Efficient planarity testing, J. ACM 21 (4) (1974) 549–568.

[23] S. Jones, P. Eades, A. Moran, N. Ward, G. Delott and R. Tamassia, A note on planar graph drawing algorithms, Technical Report 216, Department of Computer Science, University of Queensland (1991).

[24] M. Jünger and P. Mutzel, Maximum planar subgraphs and nice embeddings: Practical layout tools. Algorithmica 16 (1) (1996) (Special issue on Graph Drawing, G. Di Battista and R. Tamassia, eds.).

[25] M. Jünger and P. Mutzel, The polyhedral approach to the maximum planar subgraph problem: New chances for related problems, in: R. Tamassia and I.G. Tollis, eds., Graph Drawing (Proc. GD '94), Lecture Notes in Computer Science 894 (Springer, Berlin, 1995) 119–130.

[26] M. Jünger and P. Mutzel, Exact and heuristic algorithms for 2-layer straightline crossing minimization, in: Proc. Graph Drawing 1995, Lecture Notes in Computer Science 1027 (1996) 337–348.

[27] T. Kamada, Visualizing Abstract Objects and Relations, World Scientific Series in Computer Science (1989).

[28] T. Kamada and S. Kawai, An algorithm for drawing general undirected graphs, Inform. Process. Lett. 31 (1989) 7–15.

[29] G. Kant, Algorithms for drawing planar graphs, Ph.D. Thesis, Dept. Comput. Sci., Univ. Utrecht, Utrecht, The Netherlands (1993).

[30] G. Kant, A more compact visibility representation, in: Proc. 19th Internat. Workshop Graph-Theoret. Concepts Comput. Sci. (WG '93) (1993).

[31] G. Kant and H.L. Bodlaender, Planar graph augmentation problems, in: Proc. 2nd Workshop Algorithms Data Struct., Lecture Notes in Computer Science 519 (Springer, Berlin, 1991) 286–298.

[32] C.E. Leiserson, Area-efficient graph layouts (for VLSI), in: Proc. 21st Ann. IEEE Sympos. Found. Comput. Sci. (1980) 270–281.

[33] E. Nardelli and M. Talamo, A fast algorithm for planarization of sparse diagrams, Technical Report R.105, IASI-CNR, Rome (1984).

[34] A. Papakostas and I.G. Tollis, Improved algorithms and bounds for orthogonal drawings, in: R. Tamassia and I.G. Tollis, eds., Graph Drawing (Proc. GD '94), Lecture Notes in Computer Science 894 (Springer, Berlin, 1995) 40–51.

[35] A. Papakostas and I.G. Tollis, Improved algorithms and bounds for orthogonal drawings, Manuscript (1995).

[36] R. Read, New methods for drawing a planar graph given the cyclic order of the edges at each vertex, Congr. Numer. 56 (1987) 31–44.

[37] P. Rosenstiehl and R.E. Tarjan, Rectilinear planar layouts and bipolar orientations of planar graphs, Discrete Comput. Geom. 1 (4) (1986) 343–353.

[38] K. Sugiyama, S. Tagawa and M. Toda, Methods for visual understanding of hierarchical systems, IEEE Trans. Systems Man Cybernet. 11 (2) (1981) 109–125.

[39] R. Tamassia, On embedding a graph in the grid with the minimum number of bends, SIAM J. Comput. 16 (3) (1987) 421–444.

[40] R. Tamassia, G. Di Battista and C. Batini, Automatic graph drawing and readability of diagrams, IEEE Trans. Systems Man Cybernet. 18 (1) (1988) 61–79.

[41] R. Tamassia and I.G. Tollis, A unified approach to visibility representations of planar graphs, Discrete Comput. Geom. 1 (4) (1986) 321–341.

[42] R. Tamassia and I.G. Tollis, Planar grid embedding in linear time, IEEE Trans. Circuits Systems 36 (9) (1989) 1230–1234.

[43] R. Tamassia and I.G. Tollis, eds., Graph Drawing (Proc. GD '94), Lecture Notes in Computer Science 894 (Springer, Berlin, 1995).

[44] I.G. Tollis, Personal communication (1995).

[45] H. Trickey, Drag: A graph drawing system, in: Proc. Internat. Conf. on Electronic Publishing (Cambridge University Press, 1988) 171–182.

[46] W.T. Tutte, How to draw a graph, Proc. London Mathematical Society 13 (3) (1963) 743–768.

[47] L. Valiant, Universality considerations in VLSI circuits, IEEE Trans. Comput. 30 (2) (1981) 135–140.

[48] J.Q. Walker II, A node-positioning algorithm for general trees, Softw.—Pract. Exp. 20 (7) (1990) 685–705.

[49] S.K. Wismath, Characterizing bar line-of-sight graphs, in: Proc. 1st Ann. ACM Sympos. Comput. Geom. (1985) 147–152.

[50] D. Woods, Drawing planar graphs, Ph.D. Thesis, Department of Computer Science, Stanford University (1982), Technical Report STAN-CS-82-943.