On Improving Orthogonal Drawings: The 4M-Algorithm

Ulrich Fößmeier[1], Carsten Heß[2]⋆, and Michael Kaufmann[3]

[1] Tom Sawyer Software, 804 Hearst Avenue, Berkeley, CA 94710,
foessmei@tomsawyer.com
[2] Tom Sawyer Software, 804 Hearst Avenue, Berkeley, CA 94710,
chess@tomsawyer.com
[3] Universität Tübingen, Wilhelm-Schickard-Institut, Sand 13, 72076 Tübingen, Germany,
mk@informatik.uni-tuebingen.de

Abstract. Orthogonal drawings of graphs are widely investigated in the literature and many algorithms have been presented to compute such drawings. Most of these algorithms lead to unpleasant drawings with many bends and a large area. We present methods how to improve the quality of given orthogonal drawings. Our algorithms try to simulate the thinking of a human spectator in order to achieve good results. We also give instructions how to implement the strategies in a way that a good runtime performance can be achieved.

1 Introduction

Among the various layout styles orthogonal drawings play a central role in graph drawing. There are many applications like ER-diagrams and workflow visualization where orthogonal drawings are required. The quality of these drawings is usually estimated by analyzing them with respect to some cost function like the amount of used area, the number of bends and various others.

The two main categories of algorithms consist of

- *fast algorithms* (almost linear time) which guarantee a good worst-case performance but usually produce drawings of poor quality. Examples for such approaches are [19,14,1,2].
- *good algorithms* which guarantee good quality, but have a slow performance [18,8].

If we look at this situation from a practical point of view we can easily state that these two classes are disjoint. In several papers, the approaches have been experimentally compared (see e.g. [4]) and the good algorithms clearly outerperformed the fast algorithms with regard to the quality of the drawings.

⋆ Parts of this work were done during this author's stay at the Universität Tübingen.

S.H. Whitesides (Ed.): GD'98, LNCS 1547, pp. 125–137, 1998.
© Springer-Verlag Berlin Heidelberg 1998

Nevertheless, it is evident that not only the results of the fast algorithms but also of the good algorithms can be more or less drastically improved by manual modifications in most cases. In this paper, we present a framework for efficient local changes to improve the quality of any given orthogonal drawing. We demonstrate how the drawings produced by a fast algorithm or even by a good algorithm can be improved such that the result will be clearly better than the solution directly found by good approaches. After having done our automatic improvements, even an experienced layout person will have a hard time to find places for further manual improvements.

Since the algorithm improves the quality of the drawing step by step, this process can be done interactively. The user can determine (by passing a parameter) how much time he likes to spend. The more time he is willing to wait, the better the quality of the resulting drawing.

In the following we present four methods to save bends or area: Moving, Matching, Morphing and Merging. We demonstrate the power of these strategies by giving examples.

2 The Model

To construct orthogonal drawings, the drawing plane is subdivided by horizontal and vertical gridlines of unit spacing λ. The vertices of the graph are represented by rectangles of size $(w\lambda - \frac{\lambda}{2}) \times (h\lambda - \frac{\lambda}{2})$ for some positive integers w and h. They are placed such that the borders of the rectangles overlap the gridlines by $\frac{\lambda}{4}$. Such drawings have the nice property that the distance between two vertices on gridpoints with unit distance is the same as the size of the smallest possible vertex. The intersections of the gridlines with the rectangle define the set of *ports*, namely the positions where the start (end) points of the incident edges may be placed. All segments of the edges are placed on gridlines. They may cross but not overlap, and also the rectangles representing the vertices must be disjoint. See Fig. 1 for an example; the gridpoints are marked with a cross in this figure.

Orthogonal drawings naturaly support the drawing of vertices of degree at most 4; the strategy described above permits to handle vertices with a higher degree in the same natural way. One of the most appealing drawing models within this framework is the so-called `bignode`-model (see e.g. [7,3]). Here the number of vertical gridlines intersecting the rectangle for vertex v (roughly the width of v) is determined by the maximum of the number of incident edges at the top and bottom side of the rectangle. The height of v is restricted analogously. Thus every vertex is as large as it must be to allow a proper orthogonal edge distribution. Many previous approaches do not meet this criterion (e.g. `GIOTTO` [18], visibility representations (e.g. [16,13])) and might lead to drawings with unnecessarily large vertices.

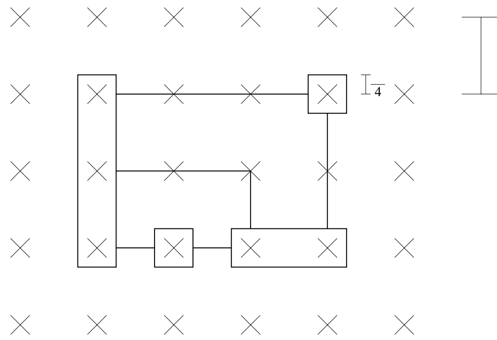

Fig. 1. Notation of the used model.

3 The 4M-Algorithm

3.1 Moving

Moving is an operation that does not change the angles and bends of the drawing but the length of edge segments. Thus Moving cannot save bends, but it is a very powerful method to save area.

The basic idea of Moving is illustrated in Fig. 2.

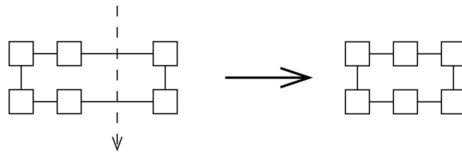

Fig. 2. A simple example for Moving.

The part of the drawing at the right side of the dashed line is moved by one unit, reducing the width of the drawing. This seems to be an easy operation, but Fig. 3 shows that even for very small graphs the situation can be more complicated.

The problem of formulating and finally implementing this method reduces to defining the possible course of the dashed line which we call *moving-line*. We perform Moving in two phases, a horizontal phase and a vertical phase. In the following we only describe the horizontal phase (the vertical phase can be formulated analogously).

Let Γ be an orthogonal drawing and Γ' the (planar) drawing where bends and edge crossings are replaced by artificial vertices. Subdivide every face of Γ' into

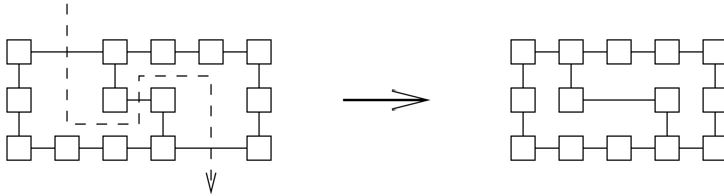

Fig. 3. A more difficult example for the moving-line.

horizontal stripes by adding artificial vertices and edges such that every stripe is a (not necessarily bounded) rectangle (see Fig. 4 for an example). The resulting drawing is called Γ'' [1].

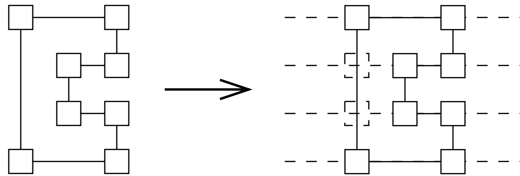

Fig. 4. An orthogonal drawing is partitioned into stripes.

The purpose for generating the drawing Γ'' is to avoid overlapping parts in the resulting drawing.

Definition 1 (moving-line J) *An area-saving moving-line J is a line that fulfils the following conditions:*

a) J is directed and consists of horizontal and vertical pieces.
b) J starts above of the topmost object of Γ'' and ends below the bottommost object of Γ''.
c) J does not intersect any vertical edge of Γ''.
d) Every horizontal edge of Γ'' that is intersected by a piece of J which is directed downward has a finite length larger than or equal to two.

After finding a moving-line J the new (smaller) drawing can be constructed by decreasing the length of every edge of Γ'' that is intersected by J in downward direction by one unit and by increasing by one unit the length of every edge of Γ'' that is intersected by J in upward direction. Since J intersects the border of Γ'' in downward direction the total area of the drawing decreases. Moving can

[1] All operations described in the rest of the paper work on Γ''. We will only describe horizontal operations.

be intuitively seen as separating the drawing in two parts (at the left side and at the right side of the moving-line) and then moving one part closer to the other one.

Finding a moving-line can be implemented as finding a path in the planar dual graph (which is well-defined since Γ'' is planar). We use dfs for that purpose.

Note that the moving-line is not necessarily monotonous in y-direction. In general it is necessary to increase the length of some edges in order to shorten other ones (see Fig. 3 for an example). For practical applications however, we often only look for monotonous moving-lines; by that, we can perform several moving-operations simultaneously, which is more efficient than in the general case (see Section 4).

Remarks:Moving resembles the compaction techniques developed for VLSI layout [10,11,12]. The three remaining operations (Matching, Morphing, Merging) are novel. The idea of dividing a drawing in two parts and modifying one part was first mentioned in [15], where one part of the drawing was turned by 90° in order to save bends.

3.2 Matching

The next operation called Matching is designed for saving bends. The idea is to save a bend b on an edge $e = (u, v)$ by moving either u or v to the place of b (to match a bend with a vertex). See Fig. 5 for an example.

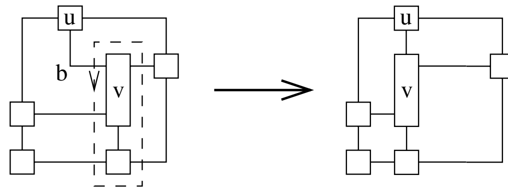

Fig. 5. Matching bend b with vertex v using a matching-line.

We use the intuition given in the previous section. Let e be an edge incident to a vertex v. Moving v to the geometric place of a bend b on e can be performed by finding a line (analogously to the moving-line) that separates the drawing in two parts, v and b being in different parts. It is not necessary to move a part of the border of the drawing because we do not want to save area. Thus the new line (the *matching-line*) J is a closed simple line that does not necessarily traverse the outer face.

Definition 2 (matching-line J) *A bend-saving matching-line J is a line that fulfils the following conditions (let s be the segment of e between v and b):*

a) J is directed and consists of horizontal and vertical pieces.
b) J starts and ends in the face at the side of e where b has its 270°-angle and intersects the segment s
c) J does not intersect any vertical edge of Γ''.
d) Every horizontal edge of Γ'' (besides s) that is intersected by a piece of J which is directed downward has a finite length larger than the length of s.

See Fig. 5 for an example. The new drawing can be constructed by shifting the whole part of the drawing Γ'' 'inside' of the matching-line to the left such that v replaces b.

3.3 Morphing

Morphing is a procedure that saves bends by resizing vertices. The basic idea of Morphing is illustrated in Fig 6.

1) Save a bend next to vertex v by expanding v such that it now covers the bend.
2) Resize the vertex back to the smallest possible size.

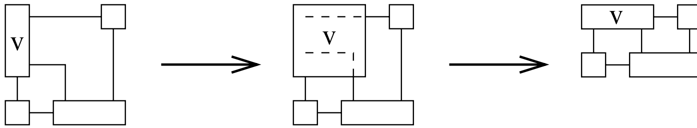

Fig. 6. The two main steps of the Morphing-algorithm.

There are three conditions why Morphing can fail in the `bignode`-model:

a) The resulting vertex intersects with another object of the graph.
b) The resulting vertex does not fit the `bignode`-condition anymore (minimal possible size). Two problems must be distinguished here:
   b1) The vertex became wider. Thus the number of edges incident to its top respectively bottom side must be checked.
   b2) The number of edges incident to the vertex' right side was decreased. Thus the height of the vertex must be checked.

Fig. 7 shows examples where Morphing is not possible because of either of the three above reasons. Note that in the middle and in the right drawing v is not a legal `bignode`.

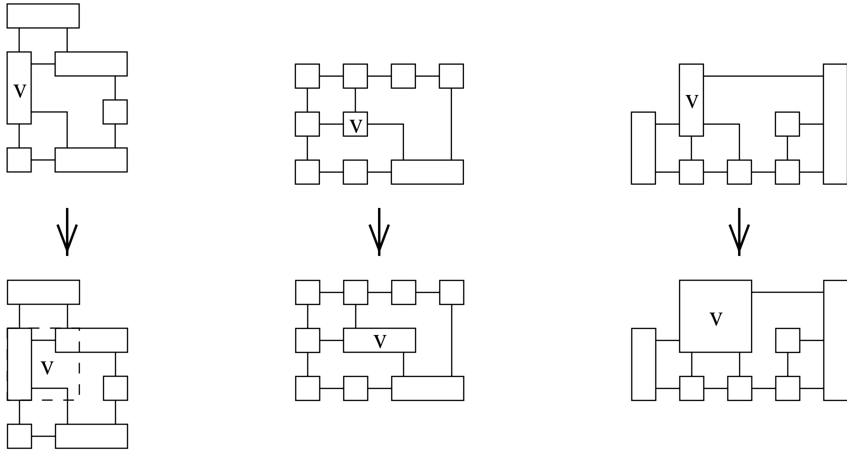

Fig. 7. Three cases where Morphing is impossible.

Let v be a vertex in an orthogonal drawing Γ'' with an incident edge e; let b be a bend on e and s the segment of e between v and b such that s has length one. Each of the three conditions can be seen as moving parts of the drawing: condition a) demands moving not the whole vertex, but moving the right border of v to the column of b. In Fig. 6 the right border of v was moved to the bend whereas the left border kept its original position. Condition b1) says that if the number of edges incident at the bottom side of v is smaller than the number of edges incident to v at its top side, then we must move the right border of v and a part of the lower border of v to the left in order to remove unused ports on both horizontal borders of v. Condition b2) says that if the number of edges incident to the left side of v is smaller than the number of edges incident to v at its right side, then we must move the lower border of v (and possibly a part of the left border, too) upward in order to remove unused ports at both vertical sides of v.

Analogously to Moving and Matching we can describe these operations with a *morphing-line*. Since we want to change the shape of a vertex we regard the vertex in question as a face and allow the morphing-line to traverse that face (resp. vertex). By that some parts of the border of the vertex are extended or shortened. The necessary conditions for the morphing-line can easily be developed analogously to Moving and Matching and are skipped here.

Fig. 8 illustrates that the morphing-line consists of three parts, all of them traversing the vertex v. Rather than regarding each part as a separate line we link the three parts together and consider them as one line. That enables us to test all three conditions simultaneously. Note that an additional Moving at the end would even save two gridlines.

Remark: The inverse operation to Morphing was described in [17], where it was used to shrink vertices by introducing bends in order to get an orthogonal drawing out of a visibility representation.

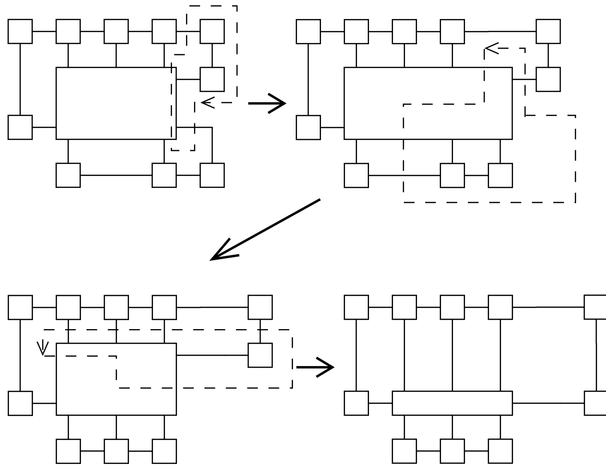

Fig. 8. The three parts of the morphing-line.

3.4 Merging

Moving tries to save area, Matching and Morphing are designed to save bends. Our last algorithm has the purpose of reduce the size of vertices. In mosts cases this also means to save area. The basic idea of this method is to merge locally two gridlines into one. Merging is a two-step procedure, Fig. 9 shows an example for a simple Merging. First we perform a Morphing in the opposite way: Vertex v is contracted by introducing a bend on the edge (v, w). This means that we have to find an inverse morphing-line to resize the vertex. In the second step this bend is matched to w. Thus the final drawing has as many bends as before, but we reduced the size of one vertex. Like for Morphing we can perform both steps simultaneously in one call of a pathfinding procedure by linking together the lines to a *merging-line* (the exact definition of this line can easily be derived from the results in the previous sections).

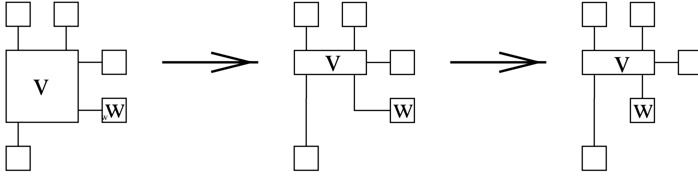

Fig. 9. An example for Merging.

4 Implementation Issues and Time Complexity

4.1 Worst Case Analysis

After having presented the ideas behind our algorithms we now analyze their runtime behavior.

Theorem 1 *The total running time of a complete run of 4M is* $O(n^2)$.

Proof (Sketch). 4M basically consists of three steps:

1. Computing the subdivision into stripes which can be done in time $O(n \cdot \log n)$ by using a sweep-line method.
2. Finding a line which takes linear time for performing a variant of dfs.
3. Redrawing the graph which is obviously a linear-time procedure.

But we do not have to recompute the data structure after each step from scratch. We maintain the stripes as balanced trees where the corresponding objects are stored. Inserting and deleting into this data structure can be done in logarithmic time, thus the time necessary before we can start to find the next line is very short. If we assume that our initial drawing only had a linear number of bends (most orthogonal drawing algorithms guarantee such a bound) we have a linear bound for the number of actions each of them requiring linear time.

4.2 Fast and Good Variants of 4M

As mentioned earlier the 4M-algorithm is designed for practical use rather than to establish new theoretical results. Running 4M without time limits leads to similar bounds of the running-time as the good algorithms themselves. But 4M can easily be implemented not to find quasi-optimal solutions, but still to find good solutions while guaranteeing a short running-time. The main observation is that almost all (matching-, morphing-, merging-) lines have a very short length

in practical examples. Thus looking for lines consisting of at most k segments and bounding k by a constant (e.g. $k = 10$) leads to almost the same results as a complete run of 4M, but only needs constant time per operation (plus updating the data structure). Moreover the parameter k can be determined by the user.

Using this strategy one matching-, morphing- or merging-step can be performed in constant time. We cannot use this technique for Moving because the moving-line has to traverse the complete drawing in one dimension; thus its length cannot be bounded by a constant. But we can be tricky here though: First we replace intermediate Movings (being necessary for example to move a bend closer to a vertex in order to perform Morphing which requires unit distance between vertex and bend) by local operations (that do not save space); these steps again can be performed in constant time. Further we restrict ourselves to do 'real' Movings at the very end of the algorithm and only compute monotonous lines. They can be computed simultaneously in a right-first dfs manner in linear time and we do not lose much quality, although being unable to find *all* possible moving-lines. Implementing all these ideas leads to algorithm `Quick-4M`:

Theorem 2 `Quick-4M` *can be implemented to run in* $O(n \cdot \log n)$ *time.*

Experiments show that the results of `Quick-4M` have almost the same quality as the drawings computed by 4M.

The main drawback of 4M is that there is no natural order how to apply the operations. Indeed the order is very important: Applying one method can destroy the necessary conditions for other improvement steps. If we are willing to spend more time we can formulate the problem as a max-flow-problem in a corresponding network, with the purpose of finding the maximum possible progress instead of any local progress. By the augmenting paths technique we can 'reverse' some actions, and find another 'order' which gives better results.

The runtime of this procedure is $O(n^2 \log n)$ which is not much worse than performing a complete 4M and we can expect to improve the quality of our drawings.

Another way of improving the algorithm is to be smart in the choice of the vertex (resp. bend) that we are dealing with. So far we try any vertex to find a suitable candidate for the starting point of a line. If we are unlucky, the 'good' candidates come at the end. To avoid this situation we can start our search at all vertices simultaneously; as soon as one line J has been found (the shortest one), we block the region inside of J from being entered by other lines. This improves the efficiency of the algorithm.

These approaches can only be hints how to improve the drawings further and require more research and experiments.

5 Conclusion

We have presented four algorithms to modify a given orthogonal drawing. Table 1 shows how these operations can change the size of the vertices, the total area of the drawing and the number of bends. A '+'-sign means an improvement with regard to the corresponding criterion, a '−'-sign a possible worsening, an '='-sign no change.

Table 1. How the methods can change the drawing.

	vertex size	area	bends
Moving	=	+	=
Matching	+ =	+ =	+
Morphing	+ = −	+ =	+
Merging	+	+	=

Even with the realization of the max-flow-network indicated in the previous section, it is still an interesting open problem to determine a good order how to perform the four methods. At least at the very end, a moving-operation should be performed because a drawing that still allows moving-steps often contradicts human users' aesthetic feeling (see e.g. Fig. 8).

The remaining question is how to compute an initial drawing for 4M. The best results can be obtained when there are not many bends on a single edge, whereas the total number of bends is not so important. Thus the staircase-algorithm (presented in [6]) is an ideal start for a drawing algorithm which uses 4M. For the staircase algorithm guarantees drawings with one bend per edge for planar graphs and the usual methods can be applied to extend it for nonplanar graphs.

Fig. 10 and 11 show an impressive example for the power of 4M: First an initial drawing computed by the staircase algorithm is shown. Then the same graph after applying 4M to the result of the staircase algorithm can be seen and the same graph drawn by the provably 'good' algorithm `Kandinsky` [7]. 4M clearly outerperforms the `Kandinsky`-result.

Summarizing the results of this paper we state that (by combining 4M with the staircase-algorithm) we have for the first time a fast algorithm that computes orthogonal drawings pleasant for a human observer. The secret of 4M is that the algorithm tries to apply exactly the operations that a person would perform intuitively. Further investigation and research in this direction are certainly necessary. For a more detailed description, practical results and a complete documentation of the implementation see [9].

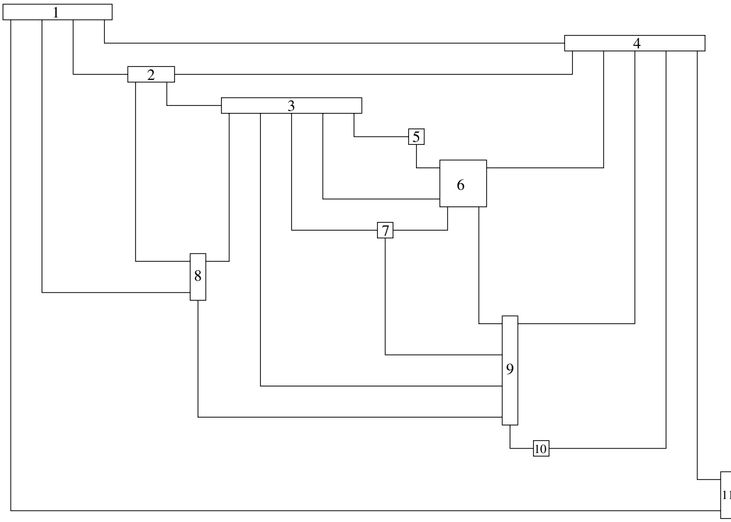

Fig. 10. An orthogonal drawing produced by the staircase algorithm.

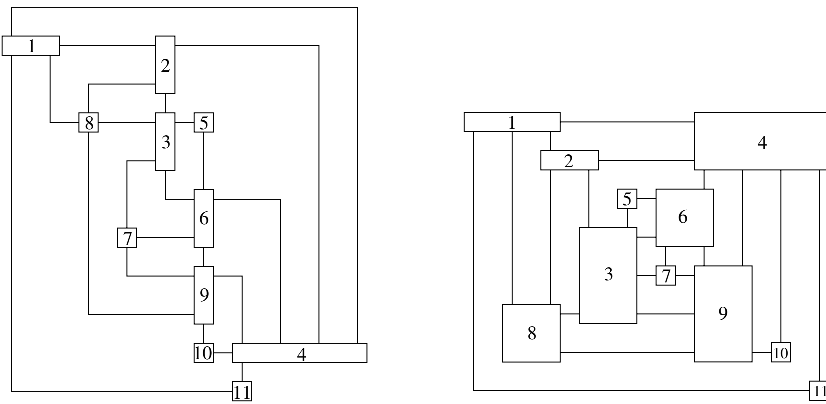

Fig. 11. Orthogonal drawings of the graph of Fig. 10 by the `Kandinsky` algorithm
(left side) and produced by 4M starting with the drawing of Fig. 10 (right side).

References

1. Biedl, T.C., G. Kant, *A Better Heuristic for Orthogonal Graph Drawings*, Proc. 2nd European Symp. Alg. (ESA'94), LNCS 855, Springer-Verlag, 1994, 124-135.
2. Biedl, T.C., M. Kaufmann, *Area-Efficient Static and Incremental Graph Drawings*, Proc. European Symp. Alg. (ESA'97), LNCS 1284, Springer-Verlag, 1997, 37-52.
3. Biedl, T.C., B.P. Madden, I.G. Tollis, *The Three-Phase Method: A Unified Approach to Orthogonal Graph Drawing*, Proc. Graph Drawing (GD'97), LNCS 1353, Springer-Verlag, 1997, 391-402.
4. Di Battista, G., A. Garg, G. Liotta, R. Tamassia, E. Tassinari, F. Vargiu, *An Experimental Comparison of Three Graph Drawing Algorithms*, Proc. ACM Symp. on Comp. Geometry, 1995, 306-315.
5. Fößmeier, U., *Orthogonale Visualisierungstechniken für Graphen*, Dissertation, Tübingen, 1997 (in German language).
6. Fößmeier, U., G. Kant, M. Kaufmann, *2-Visibility Drawings of Planar Graphs*, Proc. Graph Drawing (GD'96), LNCS 1190, Springer-Verlag, 1996, 155-168.
7. Fößmeier, U., M. Kaufmann, *Algorithms and Area Bounds for Nonplanar Orthogonal Drawings*, Proc. Graph Drawing 97, LNCS 1353, Springer-Verl., 1997, 134-145.
8. Fößmeier, U., M. Kaufmann, *Drawing High-Degree Graphs with Low Bend Numbers*, Proc. Graph Drawing (GD'95), LNCS 1027, Springer-Verlag, 1996, 254-266.
9. Heß, C., *Knicksparende Strategien für orthogonale Zeichnungen*, Diploma thesis, Tübingen, 1998 (in German language).
10. Hsueh, M.Y., *Symbolic layout and compaction of integrated circuits*, Ph.D thesis, University of California at Berkeley, 1980.
11. Mehlhorn K., S.Näher, *A Faster Compaction Algorithm with Automatic Jog Insertion* Proc. 5th MIT Conf. Advanced Research in VLSI, The MIT Press, 1988, 297-314.
12. Lengauer Th., *Combinatorial Algorithms for Integrated Circuit Layout*, Wiley-Teubner, 1990.
13. Rosenstiehl, P., R.E. Tarjan, *Rectilinear Planar Layouts and Bipolar Representations of Planar Graphs*, Discrete & Comp. Geometry, vol. 1, no. 4, 1986, 343-353.
14. Papakostas, A., I.G. Tollis, *Improved Algorithms and Bounds for Orthogonal Graph Drawing*, Proc. Graph Drawing (GD'94), LNCS 894, Springer-Verlag, 1994, 40-51.
15. Tamassia, R., *On embedding a Graph in the Grid with the Minimum Number of Bends*, SIAM J. Comput., vol. 16, no. 3, 1987, 421-444
16. Tamassia, R., I.G. Tollis, *A Unified Approach to Visibility Representations of Planar Graphs*, Discrete & Comp. Geometry, vol. 1, no. 4, 1986, 321-341.
17. Tamassia, R., I.G. Tollis, *Planar Grid Embedding in Linear Time*, IEEE Trans. Circuits Syst., vol. CAS-36, no. 9, 1990, 1230-1234.
18. Tamassia, R., G. Di Battista, C. Batini, *Automatic Graph Drawing and Readability of Diagrams*, IEEE Trans. on Systems, Man and Cybern., 18, No. 1, 1988, 61-79.
19. Tamassia, R., I.G. Tollis, *Efficient Embedding of Planar Graphs in Linear Time*, Proc. IEEE Intern. Symp. of Circuits and Systems, 1987, 495-498.