Drawing Graphs by Example Efficiently: Trees and Planar Acyclic Digraphs[*]
(Extended Abstract)

Isabel F. Cruz[1] and Ashim Garg[2]

[1] Department of Electrical Engineering and Computer Science
Tufts University
Medford, MA 02155, USA
[2] Department of Computer Science
Brown University
Providence, RI 02912–1910, USA

Abstract. Constraint-based graph drawing systems provide expressive power and flexibility. Previously proposed approaches make use of general constraint solvers, which are inefficient, and of textual specification of constraints, which can be long and difficult to understand. In this paper we propose the use of a constraint-based visual language for constructing planar drawings of trees, series-parallel graphs, and acyclic digraphs in linear time. A graph drawing system based on our approach can therefore provide the power of constraint-based graph drawing, the simplicity of visual specifications, and the computational efficiency that is typical of the algorithmic-based approaches.

1 Introduction

It is common practice to explain the layout of a graph using pictures that describe the drawing of its different components and the constraints between their relative positions. Pictures have been successfully used to convey quantitative information [28], in analogy reasoning, e.g., in geometry problems [18], for software visualization [6], and their importance in discrete mathematics has been emphasized [25]. In this paper we explore the use of pictures for expressing and computing the layout of graphs, and not only as a tool for visualizing data. In addition we show that for an important class of graphs, namely trees, and series-parallel and planar acyclic digraphs this visual computation is optimal.

In the algorithmic approach the layout of the graph is generated according to a prespecified set of general rules or aesthetic criteria (such as planarity or area minimization) that are embodied in an algorithm [12]. The algorithmic approach is computationally efficient but does not naturally support *constraints*, i.e., requirements that the user may want to impose on the drawing of a *specific* graph (e.g., clustering or aligning a given set of vertices).

Work by Tamassia *et al.* [26] has shown the importance of satisfying constraints in graph drawing systems, and has demonstrated that a limited constraint satisfaction capability can be added to an existing drawing algorithm.

[*] Research supported in part by the National Science Foundation, by the U.S. Army Research Office, and by the Advanced Research Projects Agency. Email addresses of the authors: `isabel@cs.tufts.edu`, `ag@cs.brown.edu`.

Eades and Lin [16, 23] present a system where constraints are incorporated into tree drawing algorithms and the user can fine tune the layout by giving input parameters. In both approaches each algorithm has to be individually coded. These algorithms are quite efficient because they deal with specific classes of graphs and drawings, but are difficult to extend.

Other constraint-based approaches allow declarative specification of the layout and decouple this specification from the drawing engine [19, 24], therefore requiring less expertise from the end-user than previously described techniques. However, they use general constraint solvers, which are inefficient, and constraints are textually specified using a Prolog-like syntax [19] or a set notation [24]. Even for small graphs, the enumeration of the constraints can be long and difficult to understand. Layout graph-grammars can however draw trees and series-parallel digraphs in quadratic time [5].

To avoid leaving the evaluation to a constraint solver, another possibility consists of providing the user with a tool-box of algorithmic components, which can be combined in a modular fashion. Steps have been taken in this direction by Diagram Server [13, 3], where algorithmic components are placed in an object-oriented inheritance network with coarse granularity (e.g., the algorithms for drawing trees are represented by a single node). In this model the execution of an algorithm corresponds to a path in the network. Cruz *et al.* [11], propose the combination of a declarative specification of graph layout with the tool-box approach of Diagram Server. The particular approach resides on the existence of a visual language and of a compiler that translates the visual specifications into a drawing algorithm synthesized from a database of drawing algorithms. The main purpose of the compiler is to deduce from the specifications a combination of algorithms that solves the layout problem as efficiently as possible.

As a first step towards providing a facility for writing visual programs for drawing graphs, in this paper we show how to write visual programs for constructing planar drawings of trees, series-parallel digraphs, and acyclic digraphs using the visual language DOODLE [8, 9, 10]. The drawings that we consider are: upward planar drawings of trees, Δ-drawings of series-parallel digraphs, and visibility and planar upward polyline drawings of planar acyclic digraphs. We also show that these programs construct the drawings in linear time. Thus, our approach combines the benefits of the declarative and algorithmic approaches, namely, the flexibility of the former and the efficiency of the latter.

DOODLE is a declarative rule-based language for querying database objects visually. The objects can be displayed in a variety of formats that include graphs. The similarity between the visual specification of the layout and the pictures that depict the database objects provide a "by example" approach to graph drawing. In addition, the user does not have to specify the drawing of each object in the database. Instead, drawings can be specified for database classes.

Naturally, the question arises whether algorithms and constraints expressed by a visual language such as DOODLE can be efficiently evaluated. The general inference mechanism of rule-based query languages such as Datalog [29] and F-logic [22] is a least-fixed point and bottom-up approach: starting with the database facts, rules are applied until no new facts are generated. In general this approach is inefficient, and much effort has been put in devising efficient strategies for certain classes of programs (e.g., linear Datalog programs), or in combining the bottom-up and top-down approaches [29]. However we cannot use directly the strategies devised for Datalog, because we use an object-oriented model whereas Datalog uses the relational model. In addition, DOODLE programs express constraints. Work in Datalog with constraints and the study of the time-complexity of such programs concentrates on the "bigger picture", such as characterizing which programs are in PTIME [20]. Object-oriented work for visual programming such as ThingLab [4] is also concerned with the efficient evaluation of constraints, but does not focus on graphs or other specific display classes.

This paper is organized as follows. In Section 2 we describe the class of DOODLE programs that can be executed in linear time. In Sections 3, 4, and 5 we give visual (DOODLE) programs belonging to this class for drawing trees, series-parallel digraphs, and directed acyclic digraphs in linear time. Open problems are discussed in Section 6.

2 The Computational Model

In this section we describe the evaluation strategy of DOODLE. Theorem 4 describes the DOODLE programs that can be executed in linear time. Section 2.2 gives the evaluation strategy of DOODLE to execute such programs in linear time.

2.1 Orderable and Diagonally Computable sets

In this section we introduce the notion of orderable and diagonally computable sets that we use in specifying the evaluation strategy of DOODLE.

The operators provided in DOODLE are the usual arithmetic and relational operators, and the operators **min** and **max**. Let S be a system of constraints consisting of variables $x_1, x_2, \ldots x_m$ with initial values given for some of them, the arithmetic and relational operators, and the operators **min** and **max** such that S has at least one solution. S is called *orderable* if there is an ordering π of variables such that for every distinct i and j, if there is a constraint $x_i \odot \mathbf{max}(x_j, \ldots.)$ or $x_i \odot \mathbf{min}(x_j, \ldots)$ where $\odot$ is a relational operator, then $\pi(i) > \pi(j)$; π is called a *consistent ordering* of S. A constraint of type $x_i \odot \mathbf{max}(x_j, \ldots.)$ is called a *directed* constraint. x_i is called the *head* and x_j is called a *tail* of the constraint.

S is called *diagonally computable* if it has a consistent ordering π such that every variable x_i is diagonally dependent in π. A variable x_i is *diagonally dependent* in a consistent ordering π if either there is a constraint $x_i =$ c, where c is a constant, or there is a constraint $x_i = f(x_{j_1}, x_{j_2}, \ldots x_{j_n})$, where f is some arithmetic function of the x_{j_k}'s, such that each $x_{j,k}$ is diagonally computable and $\pi(x_{j_k}) < \pi(x_i)$. We can interpret a consistent ordering as the placement of the variables in a lower triangular matrix such that x_j gets placed in the matrix element $(\pi(i), \pi(j))$ if it occurs in a constraint $x_i = f(x_j, \ldots)$ where $\pi(i) > \pi(j)$ (consequently x_j may get placed at various places). Then it is easy to see that if S is diagonally computable then starting from the variable with the lowest number in the ordering, each x_i can be evaluated from the already computed values of the other variables occurring in the row $\pi(i)$ of the matrix.

Lemma 1. *If S is an orderable system of constraints consisting of only equalities and having at least one solution, then S is diagonally computable.*

2.2 The Evaluation Strategy

We need a few definitions first. For every DOODLE rule $P \Leftarrow Q_1, Q_2, \ldots, Q_m$, P is called the *head* of the rule and $Q_1, Q_2, \ldots, Q_m$ is called the *tail* of the rule. We say that a rule *defines* a constraint if the constraint occurs in its head. DOODLE, like any other declarative system, *applies* a rule to the objects in the data base if the conditions given in its tail are satisfied. During an execution of a DOODLE program, a constraint between some attributes $x_1, x_2, \ldots, x_m$ is *created* if as a result of applying a rule, the constraint (which occurs in the head of the rule) is added to the database.

Let D be a DOODLE program. Each execution E of D corresponds to a *constraint hypergraph* whose vertices are the attributes of the objects and whose hyperedges are constraints of D created during the execution of E. We say that D has the *constraint determination property* if for every constraint C present in the tail of a rule, the following holds: (a) C is also present in the head of some rule, and (b) C is not of the type $x = \mathbf{max}(x, y, \ldots)$.

DOODLE evaluates the values of the attributes in a DOODLE program given a set of database facts as input, by executing the following two steps:

1. *Construction of a constraint hypergraph*: First construct a constraint hypergraph of the attributes of the objects in the database.
2. *Evaluation of the attribute values*: Next compute a solution by carrying out the computation over the constraint hypergraph.

2.3 Construction of a Constraint Hypergraph

Let (D, F) be a pair where D is a DOODLE program and F is set of database facts given as input to D. The construction of a constraint hypergraph for D is done by iteratively adding to the constraint hypergraph, the new attributes and constraints that are created due to the application of rules. The process stops when no new constraints can be added. We say that (D, F) has the property of *localization* if for every rule, the set of constraints that satisfy the conditions of its tail form a connected hypergraph.

A set of constraints that satisfies some conditions necessary for applying a rule but do not contradict any other conditions specified in the tail, is called a *partially satisfying set* of the rule. A partially satisfying set whose constraints form a connected hypergraph is called a *connected partially satisfying set.*

Now assume that (D, F) is a pair that has the property of localization. For such a pair, our approach to finding when to apply a rule R is:

> Maintain a set of all the connected partially satisfying sets of R. If a constraint C is created (due to applying some rule) such that C has an attribute common with a partially satisfying set S of R, then augment S by adding C to it.

Let m be the total number of constraints created by repetitively applying the rules. Let k be total number of connected partially satisfying sets of the rules that can be augmented in the above fashion to give sets whose constraints satisfy all the conditions of the corresponding rules. (D, F) is called *largely satisfiable* if $k \geq cm$, where c is a constant independent of m.

Lemma 2. *A constraint hypergraph G of a largely satisfiable pair (D, F) can be constructed in $O(m)$ time where m is the number of constraints in G, if D has the constraint determination property.*

2.4 Evaluation of the Attribute Values

We now use the constraint hypergraph constructed in the previous step to compute the values of the attributes. It can be easily shown that if a constraint hypergraph G corresponds to a diagonally computable system consisting of a set of constraints and a set of initial values of some variables, then a directed acyclic graph G^* can be constructed from G which allows the evaluation of the values of the attributes. The evaluation is done by "moving" values from the sources of G^* towards its sinks in a manner similar to topological sorting. This observations in conjunction with Lemma 1 yields the following lemma:

Lemma 3. *Given a constraint hypergraph G of an orderable system S consisting of a set of initial values of some variables and a set of constraints having only equalities, a solution of S can be computed in linear time using G.*

Lemmas 2 and 3 give the main theorem of this section.

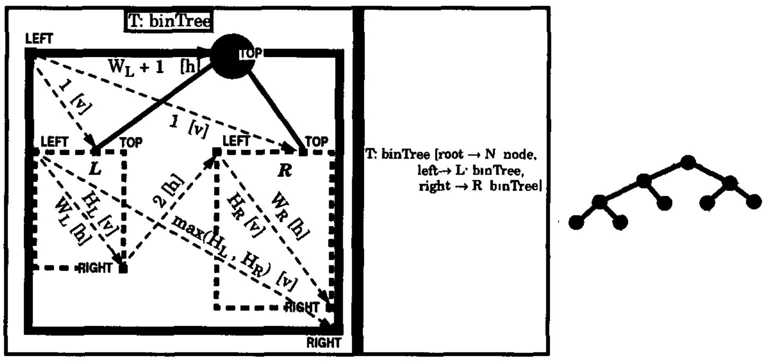
Theorem 4. *Given a* DOODLE *program D for solving an orderable system S consisting of a set F of initial values and a set of constraints having only equalities, we can compute a solution of S in linear time using D if S admits at least one solution, D has the constraint determination property, and (D, F) is a largely satisfiable pair and has the property of localization.*

3 Binary Trees

We consider planar upward straight-line drawings of binary trees such that the x- and y-coordinate of each node are proportional to the node's inorder rank and distance from the root, respectively. Also, isomorphic trees are displayed by the same layout (up to translation). This kind of layout is a variation of the □-drawing [7] (other layouts are possible by changing the position of the root).

The input to the problem is a set of objects of classes *binTree* and *leaf*, and an object of class *root*. Figure 1 shows a rule in a DOODLE program that specifies the drawing of objects of class *binTree*. The class is written on the right-hand side of the rule. The visual specification is given on the left-hand side, and consists of a circle, two lines, an invisible box, various constraints, and the drawings of the two binary subtrees represented by *reference boxes* (L and R). Each reference box recursively refers to the drawing of a binary tree given by the same rule. The invisible box (which will not get displayed) is the bounding box (that is, the tightest box) around the drawing of the tree, and contains the points between which length constraints are established.

In this rule, some of the horizontal (`[h]`) and vertical (`[v]`) length constraints are explicitly specified using dotted arrows. Other length constraints are specified using the macro constraint GRID ON: for example, when two points are on the same vertical line of the grid, then the horizontal distance between them is zero. The length constraints specify that the width of the tree's bounding box is given by the sum of the widths of the bounding boxes of the subtrees plus one, and that the height of the bounding box of the tree is the maximum height of the bounding boxes of the two subtrees plus one. An overlap constraint is also used that specifies that the circle overlaps the edges. The DOODLE rules that specify the placement of the root and of the leaves are trivial.



(i) (ii)

Figure 1: Upward layout of a binary tree: (i) DOODLE rule (ii) Layout example.

Theorem 5. *Given a binary tree T with n nodes, there exists a* DOODLE *program that constructs a planar upward straight-line drawing of T in $O(n)$ time.*

4 Series-Parallel Digraphs

A series-parallel digraph G with a source and a sink is recursively defined as follows [1]: it is either a single directed edge, or a *series composition*, or a *parallel composition* of two series-parallel directed graphs G_1 and G_2. G_1 and G_2 are called the *series-parallel components of G*.

A Δ-drawing of a series-parallel graph [1] is an upward planar drawing that is bounded by a right-angled triangle whose hypothenuse joins the source and the sink and is placed vertically. Fig. 2(i–iii) depicts the geometric construction of a Δ-drawing in the base case, series composition and parallel composition. Fig. 2(iv) gives an example of a Δ-drawing. In the base case, the bounding triangle is trivially determined. In the series composition, the bounding triangle is the smallest right-angled triangle that bounds the drawing constructed by placing the Δ-drawing of one series-parallel component at the top of the other's. In the parallel composition, the Δ-drawing of the components are placed side by side such that their bounding triangles touch at a point u on the hypothenuse of the component on the left. Point u is defined recursively: it coincides with the sink for the base case, it is the u point of the bottom component for the series composition and is the u of the right component for the parallel composition.

In a parallel composition, the poles of the left digraph are diagonally translated to the right and the poles of the right digraph are translated vertically until they coincide. Also, during a parallel composition of a digraph and an edge, the edge is always placed to the right. Although we have not shown in this paper, this condition can be easily recognized and implemented as a DOODLE rule using the class information provided for the two digraphs being composed.

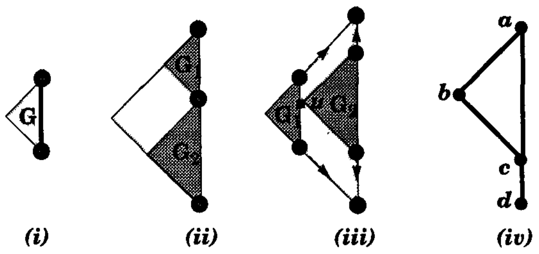


Figure 2: Geometric construction of a Δ-drawing: *(i)* Base case; *(ii)* Series composition; *(iii)* Parallel composition; *(iv)* Example of a Δ-drawing.

The DOODLE program for constructing Δ-drawings is shown in Fig. 3. We use (invisible) bounding boxes whose dimensions are appropriately set with length constraints so as to serve the same role as the bounding triangles.

Theorem 6. *Given a series-parallel digraph G with n vertices, there exists a* DOODLE *program that constructs a Δ-drawing of G in $O(n)$ time.*

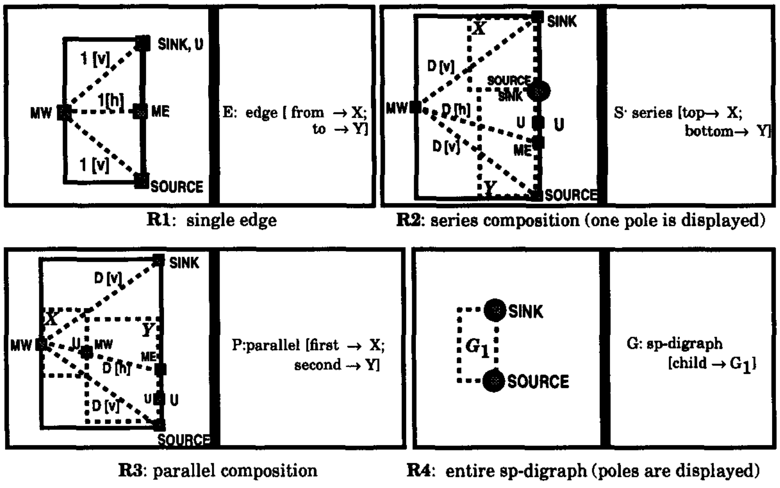

1 [v]
1[h]
1 [v]
SINK, U
MW
ME
SOURCE
E: edge [from → X; to → Y]
R1: single edge
X
SINK
D [v]
SOURCE
SINK
D[h]
U
U
D [v]
ME
MW
Y
SOURCE
S· series [top→ X; bottom→ Y]
R2: series composition (one pole is displayed)
D [v]
SINK
X
Y
MW
U
MW
ME
D [h]
D [v]
U
U
SOURCE
P:parallel [first → X; second → Y]
R3: parallel composition
SINK
G1
SOURCE
G: sp-digraph [child → G1]
R4: entire sp-digraph (poles are displayed)

Figure 3: DOODLE program that specifies a Δ-drawing.

5 Planar Acyclic Digraphs

We now present the DOODLE programs for constructing upward planar polyline drawings and visibility representations of planar acyclic digraphs in linear time. Since each upward planar acyclic digraph can be easily transformed into a planar st-digraph (i.e., a planar acyclic digraph with exactly one source and one sink) by adding some edges [21, 14], we limit our discussion to planar st-digraphs. Let G be a planar st-digraph. We assume that the input to our DOODLE programs is an upward planar embedding of G (an embedding admitting an upward planar drawing) in which the edges of each vertex are separated into two lists of its incoming and outgoing edges respectively [2, 17].

This section is organized as follows. In Section 5.1, we first show how to visually specify, using DOODLE rules, a topological numbering of a planar *st*-digraph. In Section 5.2 we give a DOODLE program to construct the dual planar *st*-digraph of a planar *st*-graph. In Sections 5.3 and 5.4 we consider visibility and upward planar polyline drawings respectively.

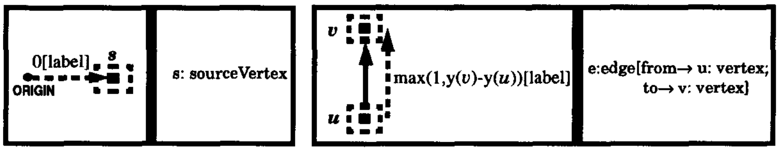
5.1 Computing a Topological Numbering

Let $G = (V, E)$ be an acyclic digraph. A *topological numbering* of the vertices of G is a mapping $l : V \rightarrow Z$ such that $l(v) > l(u)$ if there is a directed path from u to v; $l(v)$ is called the *label* of v in the topological numbering.

Figure 5.1 shows a DOODLE program D that uses the rule P given below for computing a topological numbering of a planar acyclic digraph.

$$P\text{: } l(v) = \max(l(u) + 1, l(v)) \text{ if there is a directed edge } (u, v).$$

Theorem 7. *Given a planar acyclic digraph G with n vertices, there exists a* DOODLE *program that computes a topological numbering of G in $O(n)$ time such that each source has label* 0 *and the sink has label at most n.*



R1 Set the label of source to 0 **R2**: Visual specification of Rule P

Figure 4: DOODLE program for computing a topological numbering of an acyclic digraph.

5.2 Constructing the Dual Digraph

In this section we give a DOODLE program for constructing the dual planar *st*-graph of a given planar *st*-graph G. The input to this DOODLE program is an upward planar embedding of G.

For every vertex v of G, the leftmost and rightmost edges in its list of incoming edges are called the *leftmostin* and *righmostin* edges of v. Similarly, the leftmost and rightmost in its list of outgoing edges are called the *leftmostout* and *righmostout* edges of v. The face whose boundary contains the rightmostin and righmostout edges of v is called $right(v)$ and the face whose boundary contains the leftmostout and leftmostin edges of v is called $left(v)$.

The DOODLE program for constructing the dual digraph is given in Fig. 5 and 6. Rules $R1$-$R6$ are initialization steps; $R7$ and $R8$ construct two "half" edges (f_1, e) and (e, f_2) of the desired dual edge (f_1, f_2) of each primal edge e; $R9$ constructs each dual edge by using its half edges. Attributes $Pedge_1$ and $Pedge_2$ of a vertex f in the dual graph (Fig. 5) store the primal edges of the leftmostin and righmostout edges of f respectively.

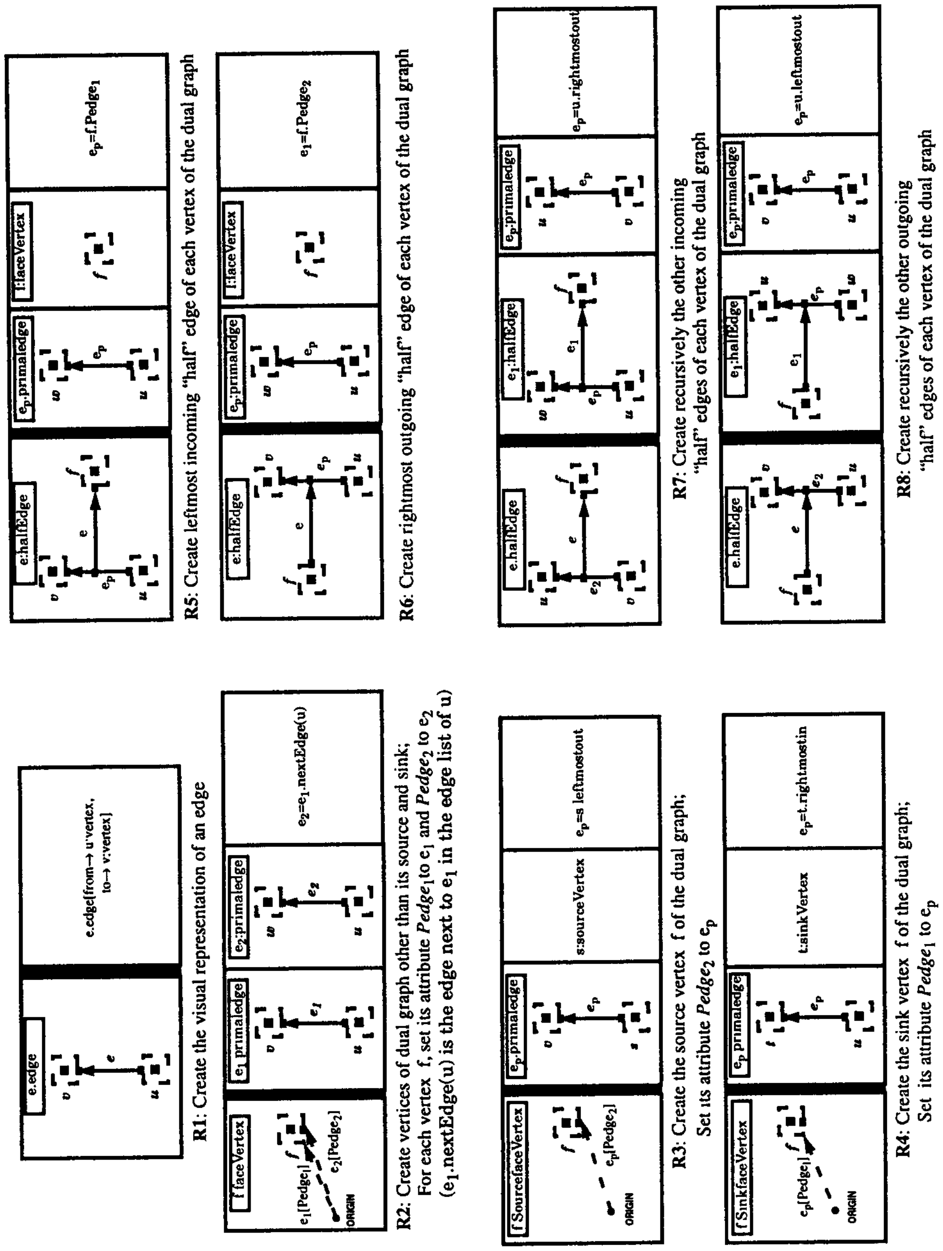
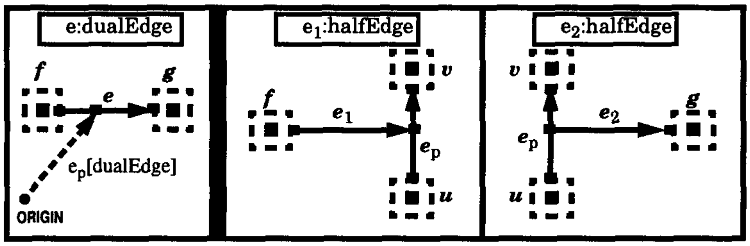
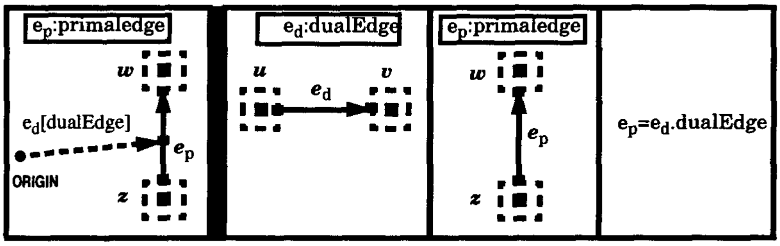

e.edge
e.edge(from→ u:vertex, to→ v:vertex)
R1: Create the visual representation of an edge
f:faceVertex
e1[Pedge1]
e2[Pedge2]
ORIGIN
e1:primaledge
e2:primaledge
e2=e1.nextEdge(u)
R2: Create vertices of dual graph other than its source and sink;
For each vertex f, set its attribute Pedge1 to e1 and Pedge2 to e2
(e1.nextEdge(u) is the edge next to e1 in the edge list of u)
f:SourcefaceVertex
ep[Pedge2]
ep:primaledge
s:sourceVertex
ep=s.leftmostout
R3: Create the source vertex f of the dual graph;
Set its attribute Pedge2 to ep
f:SinkfaceVertex
ep[Pedge1]
ep:primaledge
t:sinkVertex
ep=t.rightmostin
R4: Create the sink vertex f of the dual graph;
Set its attribute Pedge1 to ep
e:halfEdge
ep:primaledge
f:faceVertex
ep=f.Pedge1
R5: Create leftmost incoming "half" edge of each vertex of the dual graph
e:halfEdge
ep:primaledge
f:faceVertex
e1=f.Pedge2
R6: Create rightmost outgoing "half" edge of each vertex of the dual graph
e:halfEdge
e1:halfEdge
ep:primaledge
ep=u.rightmostout
R7: Create recursively the other incoming
"half" edges of each vertex of the dual graph
e:halfEdge
e1:halfEdge
ep:primaledge
ep=u.leftmostout
R8: Create recursively the other outgoing
"half" edges of each vertex of the dual graph

Figure 5: Rules $R1$ – $R8$ of the DOODLE program for constructing the dual planar st-graph of a given planar st-graph.



R9: Create each dual edge by using its half edges;
For each dual edge e, set its attribute *dualEdge* to e_p where e_p is the primal edge of e



R10: For each primal edge e, set its attribute *dualEdge* to e_d where e_d is the dual edge of e

Figure 6: Rules $R9$ and $R10$ of the DOODLE program for constructing a dual planar st-graph of a given planar st-graph.

Theorem 8. *Given an upward planar embedding of a planar st-graph G, there exists a* DOODLE *program that constructs the dual planar st-graph of G in linear time.*

5.3 Visibility Representations

A visibility representation maps vertices to horizontal line segments and edges to vertical line segments [27]. The DOODLE program consists of visually specifying each step of the following algorithm given in [27]:

1. Construct a topological numbering l of the vertices of G. Set $y(v)$ to $l(v)$.
2. Construct a dual planar directed acyclic graph G^* of G.
3. Construct a topological numbering l^* of the vertices of G. Set $x(v)$ to $l^*(v)$.
4. Draw each vertex-segment $\Gamma(v)$ at ordinate $y(v)$ and between abscissae $x(left(v))$ and $x(right(v)) - 1$.
5. Draw each edge-segment $\Gamma(e)$ where $e = (u, v)$ at abscissa $x(left(e))$ and between ordinates $y(u)$ and $y(v)$. $left(e)$ is defined as the face left of e in G.

Theorem 9. *Given an upward planar embedding of a planar st-graph G, there exists a* DOODLE *program that constructs a visibility representation of G in linear time.*

5.4 Upward Planar Polyline Drawings

The DOODLE program to construct an upward planar polyline drawing of a planar st-graph G is based on the dominance drawing algorithm presented in [15].

Theorem 10. *Given an upward planar embedding of a planar st-graph G, there exists a* DOODLE *program that constructs a planar upward polyline drawing of G in linear time.*

6 Conclusions and Future Work

In this paper we have explored the use of pictures for expressing the layout of graphs. We have shown how to perform visual layout computations in optimal time for an important class of graphs, namely trees, and series-parallel and planar acyclic digraphs. Possible future work includes:

- Visual specification of layouts for undirected graphs: Our method exploits the acyclic structure of the input digraphs to achieve optimal time complexity. We are investigating the extension of our technique to undirected graphs.
- Visual specification of global constraints: In addition to local constraints, we are investigating how to express with DOODLE aesthetic criteria associated with optimization problems (e.g., crossing or area minimization).
- A graph-drawing system based on visual specification: We envision using DOODLE as a front-end to a graph drawing system similar to Diagram Server [13] to provide at the same time the power and simplicity of visual specifications and the computational efficiency of algorithmic components.

Acknowledgements We are indebted to Roberto Tamassia for useful discussions, which contributed to the results of this paper.

References

1. P. Bertolazzi, R. F. Cohen, G. Di Battista, R. Tamassia, and I. G. Tollis. How to Draw a Series-parallel Digraph. In *Proc. 3rd Scand. Workshop Algorithm Theory, Lecture Notes in Computer Science*, vol. 621, pages 272–283. Springer-Verlag, 1992.
2. P. Bertolazzi and G. Di Battista. On Upward Drawing Testing of Triconnected Digraphs. In *Proc. 7th Annu. ACM Sympos. Comput. Geom.*, pages 272–280, 1991.
3. P. Bertolazzi, G. Di Battista, and G. Liotta. Parametric Graph Drawing. Technical Report 6/67, Consiglio Nazionale delle Ricerche, Rome, Italy, July 1992.
4. A. Borning. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.
5. F. Brandenburg. Layout Graph Grammars: the Placement Approach. In *Graph-Grammars and their Application to Comp. Sc.*. LNCS 532, Springer Verlag, 1991.
6. M. Brown, J. Domingue, B. Price, and J. Stasko, editors. *ACM SIGCHI' 94 Workshop on Software Visualization*, Boston, MA, April 1994.
7. R. F. Cohen, G. Di Battista, R. Tamassia, and I. G. Tollis. A Framework for Dynamic Graph Drawing. *SIAM J. Comput.*, to appear.
8. I. F. Cruz. DOODLE: A Visual Language for Object-Oriented Databases. In *ACM-SIGMOD Intl. Conf. on Management of Data*, pages 71–80, 1992.
9. I. F. Cruz. User-defined Visual Query Languages. In *IEEE Symposium on Visual Languages (VL '94)*, 1994.

10. I. F. Cruz. Expressing Constraints for Data Display Specification: A Visual Approach. In V. Saraswat and P. V. Hentenryck, editors, *Principles and Practice of Constraint Programming*, pages 443–468. The MIT Press, 1995.
11. I. F. Cruz, R. Tamassia, and P. Van Hentenryck. A Visual Approach to Graph Drawing. In *Graph Drawing '93*, Sèvres, France, September 1993.
12. G. Di Battista, P. Eades, R. Tamassia, and I. Tollis. Algorithms for Drawing Graphs: an Annotated Bibliography. Tech. report, Dept. of Comp. Sc., Brown University, March 1993. To appear in *Comp. Geometry: Theory and Applications.*
13. G. Di Battista, A. Gianmarco, G. Santucci, and R. Tamassia. The Architecture of Diagram Server. In *Proc. of IEEE Workshop on Visual Languages*, 1990.
14. G. Di Battista and R. Tamassia. Algorithms for Plane Representations of Acyclic Digraphs. *Theoret. Comput. Sci.*, 61:175–198, 1988.
15. G. Di Battista, R. Tamassia, and I. G. Tollis. Area Requirement and Symmetry Display of Planar Upward Drawings. *Discrete Comput. Geom.*, 7:381–401, 1992.
16. P. Eades and T. Lin. Algorithmic and Declarative Approaches to Aesthetic Layout. In *Graph Drawing '93*, Sèvres, France, September 1993.
17. A. Garg and R. Tamassia. On the Computational Complexity of Upward and Rectilinear Planarity Testing. *Graph Drawing '94* (DIMACS workshop on Graph Drawing), 1994.
18. J. G. Greeno. Conceptual Entities. *Mental Models*, D. Gentner and A. L. Stevens, ed., Lawrence Erlbaum Associates, Hillsdale, N.J., 1983, pp. 227–252
19. T. Kamada. *Visualizing Abstract Objects and Relations – A Constraint-Based Approach.* World Scientific, Singapore, 1989.
20. P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz. Constraint Query Languages. Technical Report CS-90-31, Dept. of Comp. Sc., Brown University, November 1990.
21. D. Kelly. Fundamentals of Planar Ordered Sets. *Discrete Math.*, 63:197–216, 1987.
22. M. Kifer, G. Lausen, and J. Wu. Logic Foundations of Object-Oriented and Frame-Based Languages. Technical Report 90/14 (2-nd revision), Department of Computer Science, SUNY Stony Brook, 1990. To appear in *JACM.*
23. T. Lin. *A General Schema for Diagrammatic User Interfaces.* PhD thesis, Department of Computer Science, University of Newcastle, Australia, 1993.
24. J. Marks. A Formal Specification for Network Diagrams That Facilitates Automated Design. *Journal of Visual Languages and Computing*, 2:395–414, 1991.
25. I. Rival. Reading, Drawing, and Order. In I. G. Rosenberg and G. Sabidussi, editors, *Algebras and Orders*, pages 359–404. Kluwer Academic Publishers, 1993.
26. R. Tamassia, G. Di Battista, and C. Batini. Automatic Graph Drawing and Readability of Diagrams. *IEEE Trans. on Sys., Man and Cyber.*, 18(1):10–21, 1988.
27. R. Tamassia and I. G. Tollis. A Unified Approach to Visibility Representations of Planar Graphs. *Discrete Comput. Geom.*, 1(4):321–341, 1986.
28. E. R. Tufte. *The Visual Display of Quantitative Information.* Graphics Press., Cheshire, Conn., 1983.
29. J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume II. Computer Science Press, Inc., Rockville, Maryland, 1989.