LINEAR TIME ALGORITHM FOR LAYOUT OF ACYCLIC DIGRAPHS WITH VARIABLE-SIZED VERTICES

Ari Juutistenaho

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF TAMPERE

REPORT A-1994-7

UNIVERSITY OF TAMPERE
DEPARTMENT OF COMPUTER SCIENCE
SERIES OF PUBLICATIONS A
A-1994-7, JULY 1994

LINEAR TIME ALGORITHM FOR LAYOUT OF ACYCLIC DIGRAHPS WITH VARIABLE-SIZED VERTICES

Ari Juutistenaho

University of Tampere
Department of Computer Science
P.O. Box 607
FIN-33101 Tampere, Finland

ISBN 951-44-3603-2
ISSN 0783-6910

Linear Time Algorithm for Layout of Acyclic Digraphs with Variable-sized Vertices

ARI JUUTISTENAHO
Department of Computer Science, University of Tampere
email: ari@cs.uta.fi

ABSTRACT

We investigate the problem of representing acyclic digraphs in the plane so that all edges flow from the bottom to the top and the vertices can have variable sizes. This paper includes a linear time algorithm for such a layout.

1. Introduction

Many graph drawing algorithms [DT, HL] draw the edges as straight-line segments and the vertices as points, constant-sized circles or rectangles. In practice we often have to lay out graphs that contain variable-sized vertices. We know algorithms that construct such layouts for trees [Bl, J] or for undirected graphs [N], drawing vertices as variable-sized rectangles. In this paper, we show a linear time algorithm for such a layout of acyclic digraph.

Directed graphs, *digraphs*, are used as modelling tools in several areas and therefore the problem of automatically generating plane representations of graphs has many applications. Acyclic digraphs are widely used to represent hierarchic structures. Examples include PERT networks, subroutine-call graphs, family trees, organization charts, Hasse diagrams, and ISA hierarchies in knowledge representation diagrams. A usual way to visualize the hierarchic structure of these graphs is to draw them so that all the edges follow a common general direction, e.g., from bottom to top. This concept is formalized by defining a *monotonic drawing* as a planar drawing of a digraph such that all edges are curves monotonically increasing in the vertical direction.

Our work is mainly based on the results of Di Battista and Tamassia [DT], who handle digraphs with point-sized vertices. They define three plane representations for acyclic digraphs with the above monotonic property by imposing further constraints on the plane representations for undirected planar graphs. When using planar straight-line drawings, we impose that the projection of each edge on the vertical axis is positive, and thus we get an *upward drawing*. When we combine the definitions of grid drawing and monotonic drawing, we get a *monotonic grid drawing*. In the case of visibility representations, we add the constraint that each edge-segment is directed from the lower to the higher endpoint, and call this representation of digraphs a *directed visibility representation*. The three representations of the same digraph are shown in Figure 1.

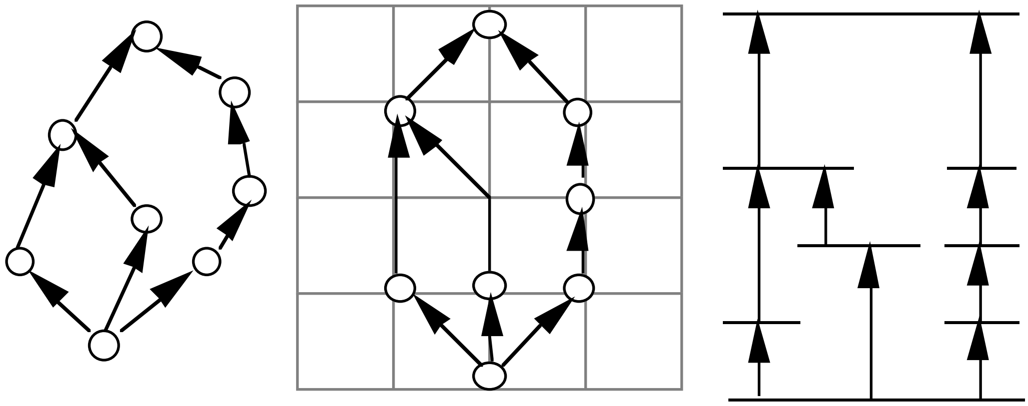

Figure 1: Upward drawing, monotonic grid drawing, and directed visibility representation

Because we handle digraphs with variable-sized vertices, those three ways to represent a digraph are not equally suitable for our purpose. The algorithm of Di Battista and Tamassia [DT] for upward drawing is based on the idea that the vertices have no size, and we have not found any easy way to modify it for graphs with variable-sized vertices. The practice has shown that if an algorithm reserves space for a vertex vertically and horizontally, then it is easy to add to it a constraint of varying size. Because the algorithm of directed visibility representation reserves space vertically for each vertex, it seems to be the most promising alternative for us.

Because we replace the horizontal segments by boxes (rectangles) and visibility representation is sometimes called horvert-representation, we call our representation of digraphs a *directed boxvert-representation*, which is demonstrated in Figure 2.

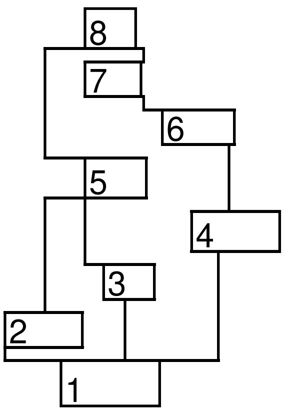

Figure 2: Directed boxvert-representation

The rest of this paper is organized as follows: Section 2 contains formal definitions and preliminary results. In Section 3, we show that the canonical numbering of the undirected graph cannot always be used as the topological numbering of the correspondent acyclic digraph, and we develop an efficient algorithm for constructing directed boxvert-representations. In the following sections, we describe the algorithm in greater detail. In Section 4, we present the algorithm for topological numbering and maximalization. In Section 5, the dual graph of the maximal acyclic digraph is constructed, and in Section 6, the coordinates for the vertices and the edges are calculated. Section 7 contains the conclusions. A C version of the algorithms is represented in the appendices. Appendices 1, 2, 3, and 4 contain C versions of the algorithms of sections 3, 4, 5, and 6, respectively.

2. Basics

For the basic graph-theoretical concepts we refer to Harary [H]. Let $G = (V, E)$ be an acyclic digraph with n vertices. A bijection $g: V \rightarrow \{1,...,n\}$ is a *topological numbering* for the vertices of G if each edge is directed from a lower-numbered vertex to a higher-numbered one.

A *directed boxvert-representation* for G consists of mapping each vertex v of G into a rectangle with upper horizontal segment $u(v)$ and lower horizontal segment $b(v)$, and each edge (v, w) into a vertical segment $(u(v), b(w))$ that has the lower endpoint on $u(v)$, the upper endpoint on $b(w)$, and does not intersect any other vertex-segments $b(u)$, $u(u)$, $u \neq v, w$.

A digraph G is a *planar st-graph* if
(1) G is a acyclic digraph;
(2) G has exactly one source, s, and one sink, t; and
(3) G contains the edge (s, t).

Di Battista and Tamassia [DT] show that the following statements are equal for any digraph G:
(1) G is a subgraph of a planar st-graph;
(2) G admits an upward drawing;
(3) G admits a directed visibility representation.

If not otherwise stated, we suppose that all graphs considered in this paper are planar st-graphs.

Garg and Tamassia [GT] have shown that upward planarity testing is an NP-complete problem. However, Bertolazzi et al. [BDMT] show a linear time algorithm for upward testing of single-source digraphs. Di Battista and Tamassia [DT] show an algorithm that constructs the directed visibility representation of such digraph in linear time.

The *dual* of G is the digraph G^* having the following properties:
(1) vertices of G^* are the faces of G, where the faces to the right and left of edge (s, t) are denoted s^* and t^* respectively;
(2) there is an edge (f, g) in G^* if face f shares an edge $(v, w) \neq (s, t)$ with face g, and face f is on the left side of (v, w), when (v, w) is traversed from v to w;

(3) finally, G^* contains the edge (s^*, t^*).

The *distance* of a vertex v, marked $a(v)$, is the length of the longest path in G from s to v.

3. Drawing Algorithm

Nummenmaa [N] has developed an algorithm that constructs a compact rectilinear planar layout for an undirected planar graph, using canonical representation. His algorithm is easy to modify for variable-sized vertices. If we could find such a canonical numbering that would be also a topological numbering, we could use his algorithm for acyclic digraphs. Unfortunately, we cannot always use that approach for our purpose. The reason is expressed in Lemma 3.1 and demonstrated in Figure 3.

Lemma 3.1. There exists a maximal acyclic digraph $G = (V, E)$ and a maximal undirected graph $G' = (V, E)$ such that no canonical numbering of G' can be the topological numbering of G.

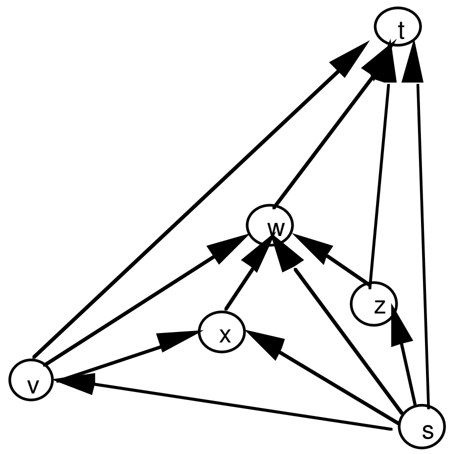

Figure 3: An example acyclic digraph G

Proof. Let G be the acyclic digraph of Figure 3. If v is a vertex of G, let $c(v)$ be the canonical number of v and let $t(v)$ be the topological number of v. It is obvious that $t(s) = 1$ and $t(t) = 6$. If we want to construct a canonical numbering that is also a topological numbering, we must define $c(s) = t(s) = 1$. After having chosen so, the only way to construct the canonical numbering is to choose $c(s) = 1$, $c(v) = 2$, $c(x) = 3$, $c(w) = 4$, $c(z) = 5$ and $c(t) = 6$.

While constructing the topological numbering of G, we can choose $t(s) = 1$, $t(v) = 2$, $t(x) = 3$, but we cannot define $t(w) = 4$ and $t(z) = 5$ because in that case there would exist an erroneously directed edge. Therefore, if $c(s) = t(s) = 1$, then $t(w) \neq c(w)$. □

The algorithm we present here is based on Algorithm VISIBILITY_DRAW of Di Battista and Tamassia [DT, p. 188]. Both algorithms map edges into vertical segments and use dual graphs. Our algorithm maps vertices into rectangles, while their algorithm maps vertices into horizontal segments. Besides, we maximalize the original graph before constructing the dual graph. When we do that, the layout is not as narrow as possible, but it is simpler to construct the dual of the graph.

As input the algorithm requires the minimum horizontal and vertical distances between the boxes (*dx* and *dy*), and the name, the width, the height, and the list of the outgoing edges for each vertex to be considered. The edges must be drawn in the same order as they are given.

The algorithm for directed boxvert-representations consists of three phases. The first phase handle the original graph so that it is easy to construct the dual graph and calculate the coordinates. It finds the sink and the source of the digraph, defines the topological numbering for the vertices, and finally maximalizes the graph. The second phase constructs the dual of the graph. The third phase calculates the values for segments of edges and for rectangles of vertices, by using the *critical path method* [E, pp. 138 - 142], which defines the distances both for the vertices in the graph and for the vertices in the dual of the graph. In what follows, we describe the phases in greater detail.

4. Topological numbering and Maximalization

We can update the outdegrees and the indegrees of the vertices already when reading the data of edges. After reading the input, we have to visit each vertex only once, to find the sink and the source of *G*.

When we construct the topological numbering, we take a vertex v that has no unvisited incoming edges and mark nr(v) = 1 + the count of vertices that have been visited before v and then we visit all the outgoing edges of v. The first vertex we visit is the source, and the last vertex is the sink.

Before we begin to maximalize *G*, we add the edge *(s, t),* if it does not already exist, to certificate that the maximalized graph is a st-graph. Read [R] shows an algorithm that maximalizes an undirected graph in linear time. It is obvious that Read's algorithm does not work correctly in our case because we have stored only outgoing edges. This is demonstrated in Figure 4. In maximalization, the vertices are visited in the order of their topological numbering. While visiting the vertex *s*, the auxiliary edges from *d* to *c* and from *e* to *d* are added. While visiting the vertex *d,* the auxiliary edge from *c* to *t* is added. The auxiliary edge from *e* to *g* should be added, but that is not done because only the incoming edges are known. Of course, we could demand that incoming edges should be given as input, but it is not necessary because we can maximalize the digraph in another way, still using linear time.

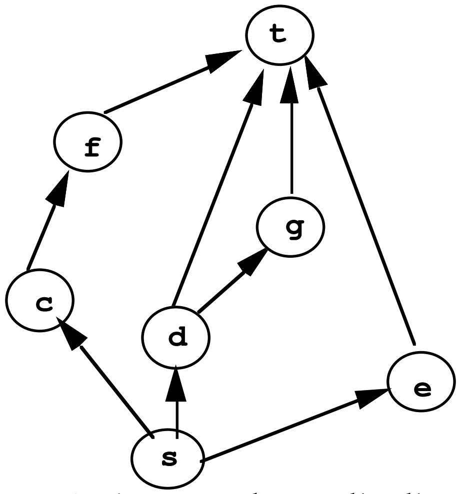

Figure 4: An example acyclic digraph

The lack of Read's algorithm is characterized in Lemma 4.1, which is demonstrated in Figure 5.

Lemma 4.1. Let G be an acyclic digraph where only outgoing edges are stored. Let G have the vertices u, w, v, v_1, .., v_n, t, and edges (u, v), (u, w), (u, t), (w, v), (w, t), (v, t), $(v, v_1,)$, .., (v, v_n). Read's algorithm does not maximalize G correctly.

Proof. While visiting u, there is a connection between v and w, and no edges are added. While visiting w, there is a connection between t and w, and no edges are added. While visiting v, there are connections between v_i and v_{i+1} and between v_n and t, and the edges $(v_1,, v_3,)$, $(v_1,, v_4)$, .., $(v_1,, v_n,)$, $(v_1,, t,)$ are added. After having applied the algorithm, there remains a face wvv_1t, and G is not totally maximalized. □

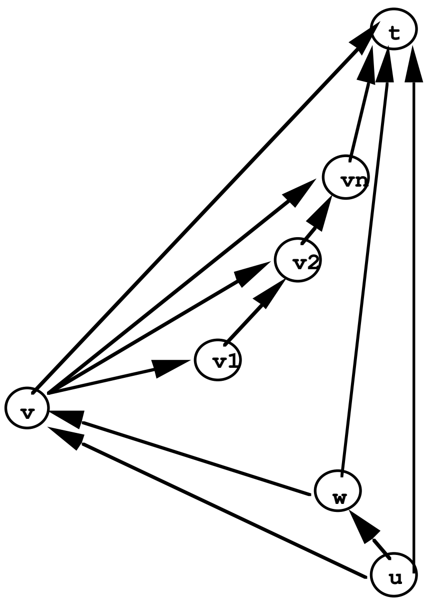

Figure 5: An example digraph G of Lemma 4.1

Di Battista and Tamassia [DT] describe a linear time algorithm for maximalization of an st-graph. For each face *f* of *G*, they define low(*f*) the vertex on the boundary of *f* with lowest topological number. Then the algorithm adds directed edges from low(*f*) to the remaining vertices of *f* nonadjacent to low(*f*). If we apply that algorithm in Figure 4, we see its weakness. While visiting the face *scftd*, the auxiliary edges are added from *s* to *f* and from *s* to *t*. While visiting the face *sdgte*, the auxiliary edges are added from *s* to *g*. But if we follow the algorithm, we either add the second edge from *s* to *f*, or the face *sdgte* remains unmaximalized.

We can solve the problems of those two algorithms by applying both of them. If we use Read's algorithm first, there remains some unmaximalized faces that can be maximalized by the other algorithm. What is more, by using Read's algorithm in the beginning, we can avoid the situations where double edges should be added.

5. Constructing the Dual Graph

Because we handle a maximalized digraph *G* that is more regular than other digraphs, it is easier for us to construct the dual graph for it. An example of a maximalized digraph with its dual is shown in Figure 6.

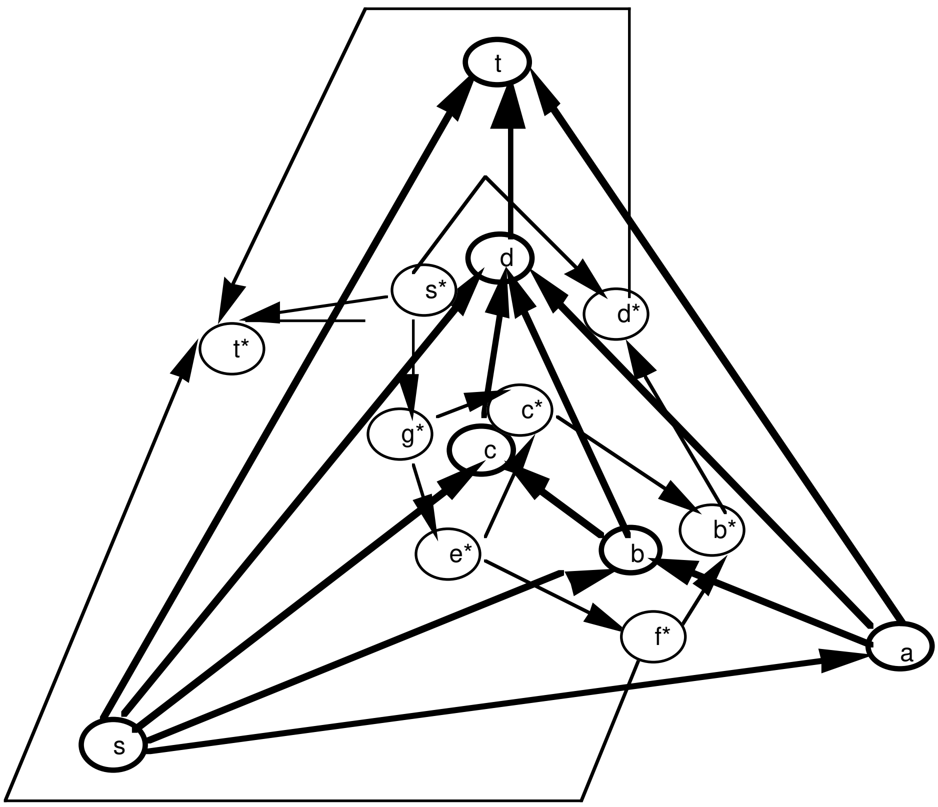

*Figure 6: An maximalized digraph G and its dual G**

Let G be a maximal acyclic digraph. If G has n vertices, it has $3n$-6 edges and $2n$-4 faces. Let (u, w) be an edge with left face l and right face r , and l^* and r^* be the corresponding vertices of G^* for the faces of G. We say that l^* is the *left vertex*, r^* is the *right vertex* and (l, r) is the *correspondent edge* of (u, w). If the edge (u, w) is the rightmost outgoing edge of u, we define that r^* is the *correspondent dual vertex* for u. In Figure 6, b^*, c^*, and d^* are the correspondent dual vertices of b, c, and d, respectively, and t^* is the correspondent dual vertex of a and s. The properties of G and G^* are described in Lemma 5.1.

Lemma 5.1. Let G be a maximalized acyclic digraph with n vertices. For G and G^*, we have the following properties:

(1) G^* has $2n$-4 vertices and $3n$-6 edges.
(2) Because each edge is connected to two vertices, then each vertex of G^* is connected to three edges. For s^*, all the edges are outgoing, for t^*, all the edges are incoming, and each other vertex of G^* has at least one incoming and one outgoing edge.
(3) There is always only one edge between two faces of G.
(4) There is for each edge of G exactly one correspondent edge in G^*.
(5) For each vertex of G, except t, there is one correspondent vertex in G^*.
(6) Let v be a vertex of G with k outgoing edges. When visiting v, we can construct k-1 vertices of G^* if $v \neq s$, and k vertices of G^* if $v = s$.

Proof. The statements 1 - 4 are obvious. The statement 5 holds by the definition of the correspondent vertex. The statement 6 is based on the idea that the dual vertices are constructed while visiting the vertex that has two outgoing edges in the face. While visiting the source, there is no incoming edge, and there are as much faces as outgoing edges. Each other vertex has at least one incoming edge between the leftmost and the rightmost outgoing edges, and if there are k outgoing edges, there are k-1 faces between them. □

We can construct the dual graph by Algorithm 1, which visits each vertex twice. During the first visit, the dual vertices are constructed, while the second visit directs the dual edges.

Algorithm 1. Constructing the dual of an acyclic digraph.

Input: A list VL of vertices sorted by the order of the topological numbering, with the list of edges given.
Output: A list DVL of vertices of the dual graph.

MakeDual(VL: vertex list):
Assign DVL ← empty list, and v ← first vertex of the VL.
Repeat
    Assign count ← v→e.
    If (v = source) **then**
        Assign count ← v→e + 1.

```
        If (count > 1) then
                Assign F, Left ← first edge of the edge list of v.
                Assign Right ← Left→next.
                Repeat
                        Create a new vertex NV.
                        Add NV to the end of DVL.
                        Assign Left→Right, Right→Left ← NV.
                        If  (Left points to a upper vertex than Right)
                                Assign BE ← Right→ToVertex→EdgeList;
                                BE→Left ← NV.
                        Else
                            Assign BE ← Left→ToVertex→EdgeList→Prev;
                                BE→Right ← NV.
                        Assign Right ← Right→next, Left ← Left→next.
                        Subtract 1 from count.
                Until (count < 2).
        Assign v ← next vertex of the VL.
Until all vertices are visited.

Assign s ← the edge from the source to the sink.
Change s→Left and s→Right.

Handle all the vertices of VL except the sink:
        Assign FE, LE ← v→EdgeList.
        Repeat
                Create a new edge NE.
                Assign NE and LE to each other's ContEdges.
                Assign From ← LE→Left and To ← LE→Right.
                Update From→Outdegree and To→Indegree.
                Assign NE→ToVertex ← To.
                Add NE to the end of From→EdgeList.
                LE ← LE→Next;
        Until (LE = FE).
```

End of Algorithm 1.

There are such constants *n* and *m* that Algorithm 1 visits each vertex and each edge *n* times, and it makes *m* calculations for each vertex and each edge. Thus, the time-complexity of Algorithm 1 is linear.

6. Calculating the coordinates

Even [E, pp. 138 - 142] presents a linear time algorithm that calculates the longest path from the source to each vertex, using critical paths. Di Battista and Tamassia [DT] use his method by computing for each vertex *v* of *G*, the length *a(v)*, the longest path from *s* to *v*, and for each vertex *f** of *G**, the length *b(f*)*, the longest path from *s** to *v**. The lengths *a(v)* and *b(f*)* are used for horizontal and vertical ordinates, respectively.

Even defines the length for each edge, and Di Battista and Tamassia define that each edge has the same length. In our case, the user defines *dx* and *dy*, the minimal vertical and horizontal distances between vertices, and we can use them for lengths of edges in *G* and *G**. Moreover, we have to use the heights and widths of vertices. The vertical direction is simple: we add the height of the vertex to its outgoing edges. For the horizontal direction, we must define the following:

1) The height of each vertex of *G** is initialized to zero.
2) The width of each vertex of *G* is added to the height of its correspondent vertex in *G**.

In Figure 7, we have a directed boxvert-representation of the digraph shown in Figure 6.

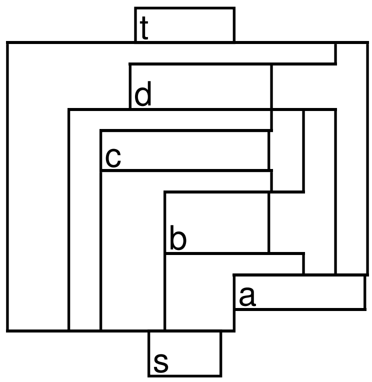

Figure 7: A directed boxvert-representation of digraph G

Figure 7 shows the reasons for the previous definition. In layout, the width of the vertex *c* affects to the edges *(b, d)*, *(a, d)*, and *(a, t)*; the width of the vertex *b* affects to the edges *(a, d)*, and *(a, t)*; and the width of the vertex *d* affects to the edge*(a, t)*. When we store the width of *v* to *v**, it affects to all edges that are located rightwards from *v* in the same horizontal level than *v* because the paths from *v** to *t** cross all those edges.

When we count the length of a vertex, we usually add the predefined minimal distance to the greatest length of the incoming edges; that distance is dy for the original graph and dx for the dual graph. There are special situations in the dual graph, where the vertex *v** in *G** is the correspondent vertex for some vertex in *G*, and has two incoming edges, like the vertices *b**, *c**, and *d** in Figure 5. We handle such vertices in a different way. Let us suppose that *u** and *w** are the vertices that have outgoing edges pointing to *v**. We make the following assignments:

a(v^*) = MAX(Length(u^*), Length(w^*));
b(v^*) = MIN(Length(u^*), Length(w^*));
Length(u^*) = dx + MAX(a(v^*), b(v^*) + Height(v^*)).

In this case, b(v^*) and a(v^*) indicate the leftmost and the rightmost ordinates, respectively, where the vertex v can be located horizontally without crossing any edges.

We can calculate the critical paths by recursive Algorithm 2. We can apply the same algorithm both to G and to G^*, by calling it with the source of the graph.

Algorithm 2. Calculating the critical paths of an acyclic digraph.

Input: The vertex VL, with the list of edges given. The fields *a*, *b*, and *z* are initialized to zero;
The list H of vertices that have no unseen incoming edges;
The boolean variable DUAL that is true if we handle G^* and false if we handle G.
Output: The fields V→a, V→z, V→b and the correspondent fields of each vertex V points to updated.

```
CriticalPaths( V: vertex; H: vertex list; DUAL: boolean):
Add V→e to V→a, V→w to V→z, dx to V→b.
If (NOT DUAL)
    Add dy to V→a.
Else (* DUAL *)
    If (V→e = 0)
        Add dx to V→a.
    Elsif (V→Indegree > 1 )
        Assign V→a ← V→a - V→e + dx.
        Assign V→a ← MAX(V→a, V→b + V→e).

If (V→Outdegree > 0)
    Handle all the edges of V:
        Assign U ← vertex points to.
        If (U→b = 0)
            Assign U→b ← V→a.
        Else
            Assign U→b ← MIN(V→a, U→b).

        Assign U→a ← MAX(V→a, U→a).
        Subtract 1 from U→UnseenEdges.

        If (U→UnseenEdges = 0)
            Add U to the end of H.
```

Assign Old ← H and H ← H→Next.
Delete(Old).

If (H is not empty)
CriticalPaths(H→V, H, DUAL).
End of Algorithm 2.

There exist such constants n and m that Algorithm 2 visits each vertex and each edge n times, and it makes m calculations for each vertex and each edge. Thus the time-complexity of Algorithm 2 is linear.

The heights we calculated during the previous phase can be directly used for the coordinates of vertices and edges. Because we want to consider some aesthetics, we calculate some values in this phase, too.

In the beginning, all the edges are visited, including the auxiliary edges that have been added during the maximalization. We define that the x-coordinate of the edge *(u, v)* = u→Right→a. Because we want to locate the edge *(s, t)* as the leftmost edge, we define exceptionally that its x-coordinate = 1. While visiting the edge *(u, v)*, the leftmost and rightmost borders for the vertices u and v are updated. After having calculated the borders, if the difference of the borders is smaller than the width of the vertex, the leftmost border is moved to the left.

To achieve a more aesthetic layout, we use Algorithm 3 to centre the vertices and to move the edges to the left if it is possible.

Algorithm 3. Centring the vertices and moving the edges of an acyclic digraph.

Input: The vertex VL, with the list of edges given. For each vertex, the fields *Xl*, *Xr*, and *w* are given; and for each edge, the field *X* is calculated.
Output: The fields V→RelX, and e→X updated.

CentreDigraph(VL: vertex list):

Handle all the vertices of V:
Assign v→RelX ← (v→Xr + v→Xl - v→w)/2.
Handle all the vertices of V:

If (v is not the sink)
Assign e1, e ← leftmost, rightmost edge of v.

While (e != e1)
Assign ep ← e→Prev and u ← e→ToVertex.
If (ep→X + dx < e→X)
Assign e→X ← ep→X + dx.

If (e→X < u→Xl)
Assign e→X ← u→Xl.
If (((e→Next→X - dx) < e→X) AND
(e→nr < e→Next→nr))
Assign e→X ← e→Next→X - dx.
If (e→X < u→RelX) AND (e→X < v→RelX)
Assign e→X ← MIN(u→RelX, v→RelX).
Assign e ← ep.

End of Algorithm 3.

It can be shown that the time-complexity of Algorithm 3 is linear.

7. Concluding remarks

The algorithm we have presented constructs the layout in linear time because each of its three phases is computed in linear time. Because the graph is maximalized anyway, the final layout is not optimal. In some cases, it could be narrower, and the edges and the vertices could be located in another way. Still, the algorithm constructs a correct layout, which can later be modified according to some aesthetics, if desired. In Figure 8 we see how our algorithm constructs a boxvert-representation of a larger acyclic digraph.

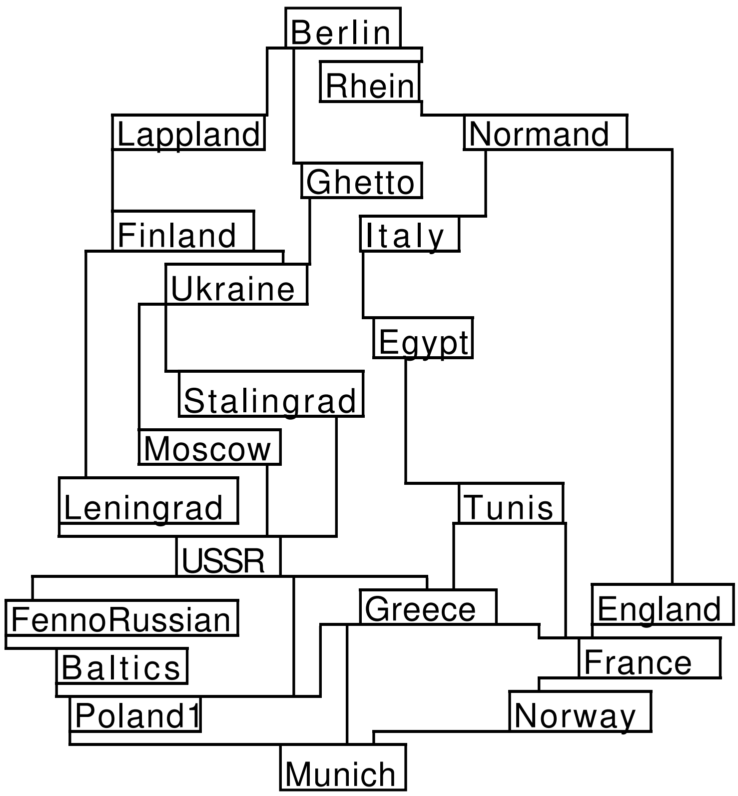

Figure 8: A directed boxvert-representation of a larger digraph

References

[BDMT] P. Bertolazzi, G. Di Battista, C. Mannino, and R. Tamassia, Optimal upward planarity testing of single-source digraphs, *1st Annual European Symposium on Algorithms (ESA '93)*.

[Bl] A. Bloesch, Aesthetic layout of generalized trees, *Software - Practice and Experience*, **23** (1993) 817 - 827.

[DT] G. Di Battista and R. Tamassia, Algorithms for plane representations of acyclic digraphs, *Theoretical Computer Science*, **61** (1988) 175 - 198.

[GT] A. Garg and R. Tamassia, On the computational complexity of upward and rectilinear planarity testing, Department of Computer Science, Brown University, Tech. Report CS-94-10, 1994.

[E] S. Even, *Graph Algorithms* (Computer Science Press, Rockville, MD, 1979).

[H] F. Harary, *Graph Theory* (Addison-Wesley, Reading, MA, 1969).

[HL] M. Hutton and A. Lubiw, Upward planar drawing of single source acyclic digraphs, unpublished manuscript, (1990).

[J] A. Juutistenaho, Linear time algorithms for layout of generalized trees, Department of Computer Science, University of Tampere, Tech. Report A-1994-6, 1994.

[N] J. Nummenmaa, Constructing compact rectilinear planar layouts using canonical representation of planar graphs. *Theoretical Computer Science* **99** (1992), 213 - 230.

[R] R. C. Read, A new method for drawing a planar graph given the cyclic order of the edges at each vertex. *Congressus Numerantium* **56** (1987), 31 - 44.

[TT] R. Tamassia and I.G. Tollis, A unified approach to visibility representations of graphs, *Discrete & Comput. Geom.* **1** (1986) 321 - 341.

APPENDIX 1: C CODE FOR SECTION 3

```
boolean         st_connected = FALSE;
edgenode        *st_connection;
int             countter = 1;
int             Sinks = 0;


boolean Digraph( VertexList )
vertexnode **VertexList;

{
     vertexnode     *Source, *Sink, *V, *W, *D, *S2, *T2;
     vertexnode     **VxArray, **Dual;
     edgenode       *ne, *fe;
     int            *UnseenVs, i, n, m, k,
                    order = 1;
     v_chainnode    *Downs, *Handled, *Handled2;
     int            *A,    /* two-dimensional array  */
                    *B, *S;    /* other helping arrays   */

     /* Allocate arrays */
     n = VertexCount;
     m = (3 * n) - 6;
     A = (int *) malloc( (n * (n - 1) / 2 + 1) * sizeof(int) );
     B = (int *) malloc( (m + 1) * sizeof(int) );
     S = (int *) malloc( (n+1) * sizeof(int) );
     VxArray = (vertexnode **) malloc( (n+1) *
                 sizeof(vertexnode *));

     /* FIRST PHASE: */

     V  = *VertexList;
     *UnseenVs = VertexCount;
     do {
          if (V->h > -1)
               FindSource(V, UnseenVs);
          V = V->Next;
     } while (*UnseenVs > 0);

     V = Source = V->Prev;     /* Let them point to the root */

     if (Sinks > 1)      /* If there are more than one sink, */
          return(FALSE); /* then we must quit */

     Sink = FindSink(V);

     Downs->Prev = Downs->Next = Downs = NewChain(Source);
     Topological(Source, Downs, order);
     SortVertices( VxArray, Source);

     if (!(st_connected)) {
          fe = Source->EdgeList->Prev;
          st_connection = TempEdge(Source, Sink, fe);
     }
     Source->EdgeList = st_connection;
```

```
        MaximDigAlt(Source, A, B, S, &k);

        for ( i = 1; i <= VertexCount; ++i) {
                VxArray[i]->z = VxArray[i]->Dout;
                VxArray[i]->a = VxArray[i]->Din;
        }

        Sink->h = 0; /* We do not handle the sink */
        MaximDig(Source, &k);

        /* SECOND PHASE: */

        for ( i = 1; i <= VertexCount; ++i) {
                VxArray[i]->Degree = VxArray[i]->Dout +
                                     VxArray[i]->Din;
                VxArray[i]->LowPt1 = VxArray[i]->Din;
                VxArray[i]->z = 0;
        }

        Dual = DualMaxDG( VxArray);
        UpdateDirs( VxArray);

        S2 = st_connection->Left;
        T2 = st_connection->Right;

        /* THIRD PHASE: */

        for (i = 1; i < VertexCount; i++)
            VxArray[i]->EdgeList->Prev->Right->e += VxArray[i]->w;

        Handled->Prev = Handled->Next = NewChain(Source);
        CriticalPaths(Source, Handled, FALSE);

        for ( i = 1; i <= VertexCount; ++i)
                VxArray[i]->a = MaxY - VxArray[i]->a;

        Handled->Prev = Handled->Next = NewChain(S2);
        CriticalPaths(S2, Handled, TRUE);

        *VertexList = Source;
        return( TRUE);
}

static void HandleDigVertices(VertexList)
vertexnode *VertexList;
{
        vertexnode     *v, *u, *up;
        edgenode  *e1, *e, *ep;
        int       counter = 0;

        v = VertexList;

        do {
                if (v->EdgeList != (edgenode *)NULL) {
                        e = e1 = v->EdgeList;
                        do {
                                e->X = (counter > 0) ? e->Right->a : 1;
                                counter++;
```

```
                    UpdateBorders(v, e);
                    e = e->Next;
               } while (e != e1);

               if ((v->Xr - v->Xl) < v->w)
                    v->Xl = v->Xr - v->w;
          }
          v->RelX = v->Xl;
          v = v->Next;
     } while (v != VertexList);

     CentreUW(VertexList);
     UpdateDownBorders(VertexList);

     v = VertexList;
     do {
          if (v->EdgeList != (edgenode *)NULL) {
               e1 = e = v->EdgeList;
               do {
                    u = e->ToVertex;
                    if (!(e->Artificial)) {
                         MoveTo(e->X, v->a);
                         LineTo(e->X, (u->a + u->e));
                    }
                    e = e->Next;
               } while (e != e1);
          }
          v = v->Next;
     } while (v != VertexList);

     v = VertexList;
     do {
          DrawBox( v->RelX, v->a, v->w, v->e );
          if (v->Name)
               DrawTrName( v->RelX, v->a, v->e, v->Name);
          if (v->Din > 0) {
               MoveTo(v->Vl, (v->a + v->e - 1));
               LineTo(v->Vr, (v->a + v->e - 1));
          }
          v = v->Next;
     } while (v != VertexList);
}


v_chainnode *NewChain( Vx )
vertexnode     *Vx;
{
     v_chainnode    *vc;
     vc = (v_chainnode *) malloc( sizeof( v_chainnode ) );

     if (vc != (v_chainnode *)NULL) {
          vc->Next = vc->Prev = (vertexnode *)NULL;
          vc->V          = Vx;
          Vx->EntryInL   = vc;
     }
     return (vc);
}
```

```
extern edgenode *TempEdge( From, To, After )
vertexnode      *From,
                *To;
edgenode        *After;

{    edgenode  *NE;     /* New Edge */

     NE = NewEdge( );

     if (NE != (edgenode *)NULL) {
          NE->Next = After->Next;
          NE->Prev = After;
          NE->Next->Prev = NE;
          After->Next = NE;
          NE->ToVertex = To;
          NE->Artificial = TRUE;

          From->Dout++;
          From->Degree++;
          To->Din++;
     }
     return (NE);
}
```

APPENDIX 2: C CODE FOR SECTION 4

```
void FindSource( V, Unseen )
vertexnode *V;
int *Unseen;
{
     edgenode       *E, *E1;
     vertexnode     *u, *v;

     V->Dout = V->Degree;
     if (V->Degree > 0) {
          E = E1 = V->EdgeList;
          do {
               u = E->ToVertex;
               u->Din++;
               u->zz++;
               if (u->h > -1)
                    FindSource( u, Unseen);
               E = E->Next;
          } while (E != E1);
     } else
          Sinks++;

     if (V->h == 0) {
          (V->h)--;
          (*Unseen)--;
     }
}
```

```
vertexnode *FindSink( Root)
vertexnode     *Root;
{
     vertexnode     *W;

     W = Root;
     do {
          if (W->Dout == 0)
               return(W);
          W = W->Next;
     } while (W != Root);
     return(W);
}

void Topological(lowest, frees, label)
vertexnode          *lowest;
v_chainnode         *frees;
int                 label;
{
     v_chainnode    *New, *Old;
     edgenode       *E, *E1;
     vertexnode     *u, *V;

     lowest->g = label;
     V = lowest;

     if (lowest->Degree > 0) {
          E = E1 = lowest->EdgeList;
          do {
               u = E->ToVertex;
               if ((u->Dout == 0) && (lowest->Din == 0)) {
                    st_connected = TRUE;
                    st_connection = E;
               }
               u->zz--;
               if (u->zz < 1) {
                    New = NewChain(u);
                    New->Prev = frees->Prev;
                    New->Next = frees;
                    frees->Prev->Next = New;
                    frees->Prev = New;
               }
               E = E->Next;
          } while (E != E1);
     }
     Old = frees;
     frees = frees->Next;
     if (Old == frees) {
          frees->V->EntryInL = (v_chainnode *)NULL;
          free(frees);
          frees = (v_chainnode *)NULL;
     } else {
          Old->Prev->Next = frees;
          frees->Prev = Old->Prev;
          Old->V->EntryInL = (v_chainnode *)NULL;
          free( Old);
     }
```

```
     if (frees != (v_chainnode *)NULL)
          Topological( frees->V, frees, label+1);
}

void SortVertices(A, F)
vertexnode     **A;
vertexnode     *F;
{
     vertexnode     *V, *Prev, *Curr, *FirstVertex;
     int  k = 0, i = 3;

     V = F;
     do {
          A[V->g] = V;
          V->nr = V->g;
          V = V->Next;
          k++;
     } while (F != V);

     /* VertexList is reordered in increasing order
      according to the canonical numbering of vertices */
     FirstVertex = Prev = A[1];
     Curr = A[2];

     Prev->Next = Curr;
     Curr->Prev = Prev;

     do {
          Prev = Curr;
          Prev->Next = Curr = A[i];
          Curr->Prev = Prev;
          i++;
     } while (i <= k);
                              /* Add last vertex */
     Curr->Next = FirstVertex;
     FirstVertex->Prev = Curr;
}
```

APPENDIX 3: C CODE FOR SECTION 5

```
void AddToList(N, L)
vertexnode     *N, **L;
{
     vertexnode *v;

     if (*L == (vertexnode *)NULL) {  /* first item */
          N->Next = N->Prev = *L = N;
     } else {
          N->Next = v = *L;
          N->Prev = v->Prev;
          v->Prev->Next = N;
          v->Prev = N;
          *L = v;
     }
}
```

```
vertexnode     **DualMaxDG(Verts)
vertexnode     **Verts;
{
     vertexnode     *LeftV, *RightV, *v;
     vertexnode     *NewV, *List;
     edgenode       *LE, *RE, *FE, *NE, *BE;
     int            i = 1, k = 1, count;

     List = (vertexnode *) malloc( sizeof(vertexnode *));
     List = (vertexnode *)NULL;
     for (i = 1; i < VertexCount; i++) {
          v = Verts[i];
          count = (i == 1) ? v->Dout + 1 : v->Dout;
          if (count > 1) {
               FE = LE = v->EdgeList;
               RE = LE->Next;

               do {
                    NewV = NewVertex((char *) NULL, 0);
                    NewV->w = NewV->e = NewV->a = 0;
                    NewV->nr = NewV->g = k;
                    AddToList(NewV, &List);
                    LE->Right = RE->Left = NewV;
                    LeftV = LE->ToVertex;
                    RightV = RE->ToVertex;

                    if (LeftV->g > RightV->g) {
                         BE = RightV->EdgeList;
                         BE->Left = NewV;
                    } else {
                         BE = LeftV->EdgeList->Prev;
                         BE->Right = NewV;
                    }
                    k++;
                    LE = LE->Next;
                    RE = RE->Next;
                    count--;
               } while (count > 1);
          }
     }

/* The direction of the edge from s* to t* must be changed: */
     FE = Verts[1]->EdgeList;
     LeftV = FE->Right;
     RightV = FE->Left;
     FE->Right = RightV;
     FE->Left = LeftV;

     return(&List);
}


void UpdateDirs(Verts)
vertexnode     **Verts;
{
     vertexnode     *From, *To, *v;
     edgenode       *LE, *FRE, *FE, *NE;
     int            i = 1, count, k;
```

```
	for (i = 1; i < VertexCount; i++) {
		v = Verts[i];
		count = v->Dout;
		if (count > 0) {
			FE = LE = v->EdgeList;
			do {
				NE = NewEdge();
				NE->nr = LE->nr;
				NE->ContEdge = LE;
				LE->ContEdge = NE;
				LE->Visited = TRUE;
				From = LE->Left;
				NE->ToVertex = To = LE->Right;
				From->Dout++;
				To->Din++;
				To->LowPt1++;
				FRE = From->EdgeList;

				if (FRE == (edgenode *)NULL)
					FRE->Next = FRE->Prev = FRE = NE;
				else {
					NE->Prev = FRE->Prev;
					NE->Next = FRE;
					FRE->Prev->Next = NE;
					FRE->Prev = NE;
				}
				From->EdgeList = FRE;
				LE = LE->Next;
			} while (LE != FE);
		}
	}
}
```

APPENDIX 4: C CODE FOR SECTION 6

```
void CriticalPaths(V, h, DUAL)
vertexnode          *V;
v_chainnode     *h;
boolean             DUAL;
{
	v_chainnode         *New, *Old, *LL;
	edgenode            *E, *E1;
	vertexnode          *u;
	int                 old;

	V->a += V->e;
	V->z += V->w;
	V->b += dx;
	if (DUAL) {
		if (V->e == 0)
			V->a += dx;
		else if (V->Din > 1) {
			V->a = V->a - V->e + dx;
			V->b += V->e;
```

```
                if (V->b > V->a)
                     V->a = V->b;
           }
     } else
           V->a += dy;

     if ((!(DUAL)) && (V->a > MaxY))
           MaxY = V->a;
     if ((DUAL) && (V->a > MaxX))
           MaxX = V->a;

     if (V->EdgeList != (edgenode *)NULL) {
           E = E1 = V->EdgeList;
           do {
                u = E->ToVertex;

                if (u->b == 0)
                     u->b = V->a;
                else if (u->b > V->a)
                     u->b = V->a;

                if (u->a < V->a)
                     u->a = V->a;
                u->LowPt1--;
                if (u->LowPt1 < 1) {
                     New = NewChain(u);
                     New->Prev = h->Prev;
                     New->Next = h;
                     h->Prev->Next = New;
                     h->Prev = New;
                }
                E = E->Next;
           } while (E != E1);
     }
     Old = h; LL = h;
     h = h->Next;

     if (Old->V->nr == h->V->nr) {
           h->V->EntryInL = (v_chainnode *)NULL;
           free(h);
           h = (v_chainnode *)NULL;
     } else {
           Old->Prev->Next = h;
           h->Prev = Old->Prev;
           Old->V->EntryInL = (v_chainnode *)NULL;
           free( Old);
           Old = (v_chainnode *)NULL;
           CriticalPaths( h->V, h, DUAL);
     }
}


static void CentreUW(VertexList)
vertexnode *VertexList;
{
     vertexnode     *v, *u, *up;
     edgenode  *e1, *e, *ep;
```

```
     v = VertexList;
     do {
          v->RelX = (v->Xr + v->Xl - v->w)/2;
          v = v->Next;
     } while (v != VertexList);

     v = VertexList;
     do {
          if (v->EdgeList != (edgenode *)NULL) {
               int a = v->RelX;
               int z = v->RelX + v->w - 1;
               e1 = v->EdgeList;
               e = v->EdgeList->Prev;

               while (e != e1) {
                    ep = e->Prev;
                    u = e->ToVertex;
                    if (ep->X + dx < e->X)
                         e->X = ep->X + dx;
                    if (e->X < u->Xl) {
                         e->X = u->Xl;
                         if ( ((e->Next->X - dx) < e->X) &&
                              (e->nr < e->Next->nr) )
                              e->X = e->Next->X - dx;
                    }
                    if ((e->X < u->RelX) && (e->X < v->RelX))
                         e->X = (u->RelX < v->RelX) ?
                                 u->RelX : v->RelX;
                    if (!(e->Artificial)) {
                         z = (e->X > z) ? e->X : z;
                         a = (e->X < a) ? e->X : a;
                    }
                    e = ep;
               }

               if (!(e->Artificial)) {
                    z = (e->X > z) ? e->X : z;
                    a = (e->X < a) ? e->X : a;
               }
               MoveTo(a, v->a);
               LineTo(z, v->a);
          }
          v->Vl = v->RelX;
          v->Vr = v->RelX + v->w - 1;
          v = v->Next;
     } while (v != VertexList);

}


static void UpdateBorders(v, e)
vertexnode     *v;
edgenode  *e;
{
     int       vertline;
     vertexnode     *u;
     vertline = e->X;
     u = e->ToVertex;
```

```
		if (v->h < 1) {
			(v->h)++;
			v->Xl = v->Xr = vertline;
		}

		if (u->h < 1) {
			(u->h)++;
			u->Xl = u->Xr = vertline;
		}

		v->Xl = (v->Xl > vertline) ? vertline : v->Xl;
		v->Xr = (v->Xr < vertline) ? vertline : v->Xr;
		u->Xl = (u->Xl > vertline) ? vertline : u->Xl;
		u->Xr = (u->Xr < vertline) ? vertline : u->Xr;
}


static void UpdateDownBorders(L)
vertexnode	*L;
{
	int	vertline;
	vertexnode	*u, *v;
	edgenode	*e, *e1;

	v = L;
	do {
		if (v->EdgeList != (edgenode *)NULL) {
			e = e1 = v->EdgeList;

			do {
				vertline = e->X;
				u = e->ToVertex;

				if (!(e->Artificial)) {
					u->Vl = (u->Vl > vertline) ?
						vertline : u->Vl;
					u->Vr = (u->Vr < vertline) ?
						vertline : u->Vr;
				}
				e = e->Next;
			} while (e != e1);
		}
		v = v->Next;
	} while (v != L);
}
```