Graphviz and Dynagraph – Static and Dynamic Graph Drawing Tools

2003

John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull

AT&T Labs - Research, Florham Park NJ 07932, USA

1 Introduction

Graphviz is a collection of software for viewing and manipulating abstract graphs. It provides graph visualization for tools and web sites in domains such as software engineering, networking, databases, knowledge representation, and bio-informatics. Hundreds of thousands of copies have been distributed under an open source license.

The core of Graphviz consists of implementations of various common types of graph layout. These layouts can be used via a C library interface, stream-based command line tools, graphical user interfaces and web browsers. Aspects which distinguish the software include a retention of stream-based interfaces in conjunction with a variety of tools for graph manipulation, and support for a wide assortment of graphical features and output formats. The former makes it possible to write high-level programs for querying, modifying and displaying graphs. The latter allows Graphviz to be useful in a wide range of areas, with applications far removed from academic exercises.

The algorithms of Graphviz concentrate on static layouts. Dynagraph is a sibling of Graphviz, with algorithms and interactive programs for incremental layout. At the library level, it provides an object-oriented interface for graphs and graph algorithms.

2 Applications

Many applications employ Graphviz to produce graph layouts in order to assist their users to better understand domain information or to perform some task visually. In particular, the stream model supported by Graphviz lends itself to applications that need an external graph visualization service with a graphical or web interface. It is simple to emit graph models in the *dot* language [15] and then load them into a customized version of one of the Graphviz viewers, or to generate server-side web content as clickable images, Adobe PDF or SVG metafiles.

We will briefly survey some successful application areas.

2.1 Software Engineering.

The complexity of large software modules, programs and protocols is a serious impediment to understanding and changing them, and thus is a problem with great economic significance. Software visualization is one attack on this problem. The idea is to model some aspect of software as a graph, and present the graph as a drawing to make it easier to understand the model. Graphs are convenient for describing the data types, functions, variables, control structures, files and even bugs in source code programs, or the structure of finite state machines and grammars. They can be created from static analysis, dynamic traces, or other sources. Some practical systems that rely on Graphviz for software visualization are the Acacia [4], Doxygen [35], and Mono [10] static analysis systems, the Syntacs toolkit for Java compiler generation [24], the Spin concurrent protocol analyzer [21] and the Bugzilla bug tracking system [1] originally created for the Mozilla (Netscape) open source project.

Graphviz has also been applied to digital logic design, database schema design, knowledge representation, Bayesian networks and decision diagrams, to name a few other areas in related branches of engineering and technology.

2.2 Bio-informatics.

Graphs arise naturally in metabolic network models, gene and protein sequences and in studies of other biological structures. Graphs are often generated from experimental data, or extracted by cross-referencing the literature. For example, PubGene [23] is a biological database application that employs Graphviz as a web visualization service. The database describes the co-citation of mouse, rat and human gene symbols in an archive of over 10 million articles from PubMed. Interactive queries allow exploring the neighborhood around a set of genes given by standard names.

The Protein Interaction Extraction System (PIES) [38] is a platform for exploring bio-medical abstracts using Graphviz. With it, the user can call up research abstracts from online sources, extract relevant information from the texts and manipulate interaction pathways. The system uses Graphviz to display interactions graphically.

The Bioconductor Graph Project (based on the R statistics language) incorporates Graphviz as a rendering module, along with other graph libraries. The integration of statistical and graph models is a promising area for data mining and visualization research.

2.3 Internet and Web Structures.

Many internet and web mapping and analysis tools are based on graphs. In the area of web structure, a central effort of the World Wide Web (W3C) consortium is to define a “semantic web” in XML. One of its contributions

if the RDF (Resource Description Framework) dialect of XML which formally describes web site contents. RDF models naturally give rise to graphs. IsaViz and FRODO RDFSViz are translators which map aspects of RDF into Graphviz diagrams. Other examples in the realm of web and internet engineering are Webdiff [5] (for tracking changes in web site contents), the Apache2Dot translator (for viewing links followed by clients within a web site), Gnucleus [7] (a visualizer for Gnutella peer-to-peer networks), DNS Bajaj [22] (for viewing and debugging domain name server delegation graphs) and netmap [34] (for traceroute visualization).

A common technique used in web pages for creating interactive content based on graphs is to rely on a *webdot* HTTP server. It is invoked by a URL which specifies a remote graph file to be retrieved, the Graphviz layout program to run, and the MIME type of the image to be created. For example, the line

```
<img src=/cgi-bin/webdot/tut1.dot.neato.png>
```

in a web page indicates that the graph described in `tut1.dot` should be drawn using *neato* and the output should be in PNG format. In addition to providing inline images, if a node or edge in the graph specifies a URL attribute, the corresponding image will act as a link to that URL. This type of web service followed naturally from the basic stream orientation of the Graphviz software.

Histograph is an application of Dynagraph that displays a nonlinear web click history graph for Microsoft Internet Explorer. In the conventional linear history view of most browsers, it is difficult or even impossible to understand branching URL visit structures. Histograph instead makes a map of the pages visited by the browser using nodes in a graph. As the user follows links in the browser, Histograph dynamically adjusts the map, and the user can easily jump to any previously explored page by clicking on its node in the map. Histograph is a concise C++ program that passes events between the Internet Explorer and Dynagraph components of the application.

Histograph was created with Montage, a generic OLE client-server module for integrating Dynagraph (or other applications) with Microsoft OLE-aware Windows programs. Montage supports user interface modes (collections of behaviors), event management and persistence of non-hierarchical collections of objects to enable state file saving and loading and cut-and-paste operations. Its generic features enable the creation of sophisticated applications. Beyond the Histograph demonstration, it supports general embeddable diagrams, and Visual Basic programming with graph diagrams.

2.4 Dynamic Distributed Communication Services.

Distributed Feature Composition (DFC) is an architecture for specifying the structured composition of modular communication service features. DFC ap-

pears to solve many of the difficulties that have been encountered in specifying telecommunication services. Of principal concern here, it models the invocation and interaction of communication services as an evolving graph of feature boxes. Building Box [2] is a platform for applying DFC to Internet Protocol services. An extension of Graphviz was created to monitor and validate service protocols and feature setups in real time. In particular, DFC models are naturally represented as a set of boundary nodes surrounding a cloud of internal feature nodes, and drawn using a modified spring embedder.

3 Algorithms

The algorithms forming the fabric of the Graphviz software range from standard graph and graph drawing algorithms, implemented for robustness and flexibility, to novel variations of standard algorithms or standard algorithms used in novel ways. It seems most natural to describe these techniques in the context of the graph drawing model where each is used, saving those serving multiple models to the end.

3.1 Static layered drawings

For layered drawings, Graphviz relies on an implementation of the Sugiyama-style approach as described in Section 4.2 of Chapter 2. As with all Graphviz drawing tools, the design goal is to make aesthetically pleasing drawings of modest-sized graphs approaching the quality of hand-made diagrams. We concentrate here on the aspects where the Graphviz implementation differs significantly from the description in Chapter 2.

Ranking. The first major pass in creating a Sugiyama-type layout is to place nodes on discrete ranks, honoring the direction of the edges. There are many ways of doing this, depending on which aspects of the ranking are deemed most important. Graphviz models node ranking as the following linear integer program:

$$minimize \sum_{(u,v)\in E} \omega(u,v)(y_u - y_v) \tag{1}$$

$$subject\ to\ y_u - y_v \geq \delta(u,v)\ for\ all\ (u,v) \in E \tag{2}$$

where y_u denotes the rank of node u and hence is a non-negative integer, and $\delta(u,v)$ is the minimum length of the edge. By default, δ is taken as 1, but the general case supports flat edges, when the nodes are placed on the same rank ($\delta = 0$), or the times when it is important to enforce a greater separation

($\delta > 1$). The weight factor $\omega(u, v)$ allows one to specify the importance of having the rank separation of two nodes be as close to minimum as possible.

Using this criterion for placing nodes on ranks has the effect of reducing the total length of all the edges where, in this context, the length of an edge is the difference in ranks of its endpoints. This is important from an aesthetic and cognitive sense, since it is generally agreed that having short edges in the drawing of a graph is important. This approach also has the practical effect of reducing the number of artificial nodes introduced for the remainder of the layout. As the time to finish the later phases is strongly influenced by the number of nodes, real and artificial, anything that reduces the number of artificial nodes needed can have a beneficial effect on performance. On the other hand, for shallow but wide hierarchies, minimizing the total edge length, or the number of ranks, can lead to a layout with a very poor aspect ratio. This can be overcome by the use of additional constraints, such as adding invisible edges between nodes which would normally be placed on the same rank.

Despite the proposed advantages of using the integer program (1-2) to determine ranks, if one could not solve it efficiently, it would not be worthwhile. Fortunately, the problem allows many polynomial-time solutions. Since its corresponding constraint matrix is totally unimodular, a rational solution obtained from a network flow formulation or the basic linear program is equivalent to the desired integer solution.

Here, we describe the network simplex algorithm used in Graphviz. We expand the notion of ranking to be any assignment of y coordinates to the nodes. A *feasible* ranking is one satisfying the length constraints (2). Given any ranking, not necessarily feasible, the *slack* of an edge is the difference of its length and its minimum length. Thus, a ranking is feasible if the slack of every edge is non-negative. An edge is *tight* if its slack is zero.

A spanning tree of a graph induces a ranking, or rather, a family of equivalent rankings. (Note that the spanning tree is on the underlying unrooted undirected graph, and is not necessarily a directed tree.) This ranking is generated by picking an initial node and assigning it a rank. Then, for each node adjacent in the spanning tree to a ranked node, assign it the rank of the adjacent node, incremented or decremented by the minimum length of the connecting edge, depending on whether it is the head or tail of the connecting edge. This process is continued until all nodes are ranked. A spanning tree is *feasible* if it induces a feasible ranking. By construction, all edges in a feasible tree are tight.

Given a feasible spanning tree, we can associate an integer *cut value* with each tree edge as follows. If the tree edge is deleted, the tree breaks into two connected components, the tail component containing the tail node of the edge, and the head component containing the head node. The cut value is defined as the sum of the weights of all edges from the tail component to the

head component, including the tree edge, minus the sum of the weights of all edges from the head component to the tail component.

Typically (but not always, because of degeneracy) a negative cut value indicates that the weighted edge length sum could be reduced by lengthening the tree edge as much as possible, until one of the head component-to-tail component edges becomes tight. This corresponds to replacing the tree edge in the spanning tree with the newly tight edge, obtaining a new feasible spanning tree. An example of this interchange is given in Figure 1. The graph has 8 nodes and 9 edges, the minimum edge length is 1, and non-tree edges being dotted. The numbers attached to tree edges are the cut values of the edge. In (a), the tree edge (g, h) has cut value -1. In (b), it is removed from the tree and replaced by edge (a, e), strictly decreasing the total edge length.

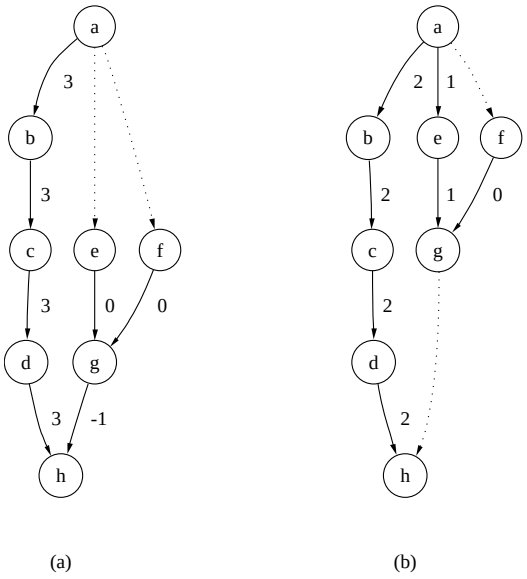

Fig. 1. A step in network simplex

It is simple to see that an optimal ranking, in the sense of the integer program (1)-(2), can be used to generate another optimal ranking induced by a feasible spanning tree. These observations are the key to solving the ranking problem in a graphical rather than algebraic context, as described in Algorithm 1. Tree edges with negative cut values are replaced by appropriate non-tree edges, until all tree edges have non-negative cut values. The resulting spanning tree corresponds to an optimal ranking.

For further discussion of the termination of the network simplex algorithm and optimality of the result, as well as implementation tricks, the interested

Algorithm 1: Network simplex

Input : Directed acyclic graph $G = (V, E)$
Output: Optimal ranking of V

Create initial feasible spanning tree T
while *edge* $e \in T$ *has negative cut value* **do**
  Pick edge $f \in E \backslash T$ with minimum slack
  Set $T = T \cup \{f\} \backslash \{e\}$
end

reader is referred to the literature, e.g., Cook et al. [9], Chvatal [6] or Gansner et al. [16].

Coordinate assignment. As noted in Chapter 2, the assignment of y coordinates for top-down drawings is basically trivial. On the other hand, picking good x coordinates in order to minimize edge bends and obtain a compact, neat layout takes some work. We attempted to use heuristics similar to those used for crossing reduction, but the heuristics became increasingly complex and started to interfere with each other. It was then recognized that we could again model node placement as a non-linear integer program:

$$\begin{aligned} &minimize \sum_{(u,v)\in E} \Omega(u,v)\omega(u,v)|x_u - x_v| \\ &\quad subject\ to\ x_a - x_b \geq \rho(a,b)\ for\ all\ a\ and\ b \\ &\quad where\ a\ is\ the\ left\ neighbor\ of\ b\ on\ the\ same\ rank. \end{aligned} \tag{3}$$

In this program, $\rho(a, b)$ gives the minimum horizontal separation of a and b, which is usually taken as the sum of half their respective widths, plus some constant internode spacing. Ω is an additional weight function favoring the straightening of long edges. Specifically, Ω is greatest where both vertices are artificial, less when only one vertex is, and least when both vertices are real.

A standard transformation in linear programming introduces additional variables to remove the absolute value. Graphically, this corresponds to creating a new graph G' as illustrated in Fig. 2. (For this presentation, we are ignoring flat edges in G.) The new graph has the same vertex set as G plus a new vertex n_e for each edge. There are two kinds of edges in G'. The first class is defined by creating two edges $e_u = (n_e, u)$ and $e_v = (n_e, v)$ for every edge $e = (u, v)$ in G. These edges have $\delta = 0$ and $\omega = \omega(e)\Omega(e)$, and thereby encode the cost of the original edge. The other type of edges separates adjacent nodes on the same rank. If v is immediately to the left of w on its rank, we add an edge $f = e_{(v,w)}$ to G' with $\delta(f) = \rho(v, w)$ and $\omega = 0$. Note that this edge will not effect the cost of the layout.

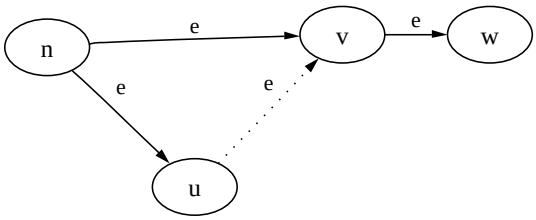

Fig. 2. Constructing G'

With this construction, solving the original optimization problem becomes equivalent to finding an optimal ranking in the derived graph G', and we can just reuse the network simplex algorithm.

This formulation has an additional advantage. By appropriately setting the minimum edge lengths, rather than using the default of 0, the derived graph can encode horizontal shifts in edge endpoints to allow node ports. This enables the drawing of arrows between fields in records, as shown in Fig. 3.

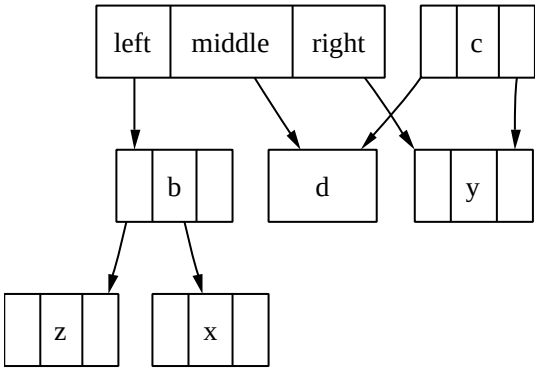

Fig. 3. Records, fields and node ports

If $e = (u, v)$ is an edge, let Δ_u and Δ_v be the desired horizontal displacements for the edge endpoints from the centers of u and v, respectively. A negative Δ corresponds to the port occurring to the left of the vertex center. We can then modify the optimization problem (3), making the cost of an edge $\Omega(e)\omega(e)|x_u - x_v + d_e|$, where $d_e = |\Delta_v - \Delta_u|$, and with $\delta(e_u) = d_e$ and $\delta(e_u) = 0$, assuming without loss of generality that $\Delta_v \geq \Delta_u$. By applying the construction for G' and using network simplex, we end up with desired horizontal coordinates and port displacements.

Edge drawing. As a final step, chains of artificial nodes are used to guide the construction of splines, which then replace them. Although we use some special techniques for layered graph, in particular to handle parallel and flat edges, the essence of the approach used in Graphviz will be described below in Section 3.4 concerning the spline path planner.

3.2 Virtual Physical Layouts

For so-called symmetric layouts,[1] Graphviz provides two algorithms using virtual physical models. One is an implementation of the Kamada-Kawai spring layout algorithm [25]. This is basically a variation on the multidimensional scaling algorithm devised in the statistics community in the 1950's and 1960's, and was first proposed as a graph layout algorithm by Kruskal and Seery [27] in 1978. In addition to the standard model using path lengths in the graph for the difference matrix, our implementation also provides a circuit model based on Kirchoff's laws suggested by Cohen [8]. This encodes the number of paths between two nodes in the distance calculation, and has the effect of making clusters tighter.

A second symmetric layout is provided which implements several of the spring-based force models described in Chapter 2. For large graphs, it relies on dynamic bins, an extension of the technique proposed by Fruchterman and Reingold [14], to approximate long distance repulsive forces, thereby reducing the running time to roughly linear. In addition, it supports hierarchical clustered graph using recursion.

Removing node overlaps. Virtual physical layout solvers usually assume that nodes are drawn as points and edges as straight lines[2] Problems arise if nodes are drawn as shapes having area, because they often overlap other nodes and edges. If the graph is large or the intention is to just see the "shape" of the graph, such node overlaps are unimportant. For small to medium graphs, however, the user typically does not want nodes occluding each other.

Graphviz has three optional strategies for removing node overlaps. One eliminates them by uniformly scaling up distances between node centers while retaining node sizes [30]. This preserves overall relationships between nodes, but can waste considerable area. A second approach is the force scan method [31] of Misue et al. Here, the layout is searched by horizontal and vertical scan passes, and rigid translations of subsets of nodes are performed in the

[1] In the vernacular, not mathematical sense.

[2] Incorporating node size into the model without introducing new problems such as overconstraining the layout is a subtle problem (cf. [19]). We have implemented some of the algorithms in the literature which include node sizes as part of the model, and have found that there are situations where overlaps can still occur.

scan direction to remove overlaps. This preserves orthogonal ordering while usually, but not always, requiring less space than scaling.

The third technique (Algorithm 2) is a more sophisticated iterative heuristic using Voronoi diagrams, based on work by Lyons et al. [29]. The rationale

Algorithm 2: Voronoi adjustment

```
Input  : Layout of vertex set V
Output: New layout of V such that v ∩ u = ∅∀v, u ∈ V

Construct bounding rectangle containing all nodes
Let C be the number of intersecting pairs of vertices
while C > 0 do
    Construct Voronoi diagram using vertex centers as sites
    Clip unbounded cells to bounding rectangle
    Move each vertex to the centroid of its cell
    Let C′ be the new number of intersections
    if C′ ≥ C then
        Expand bounding rectangle
    end
    Let C = C′
end
```

behind this technique is that moving a node in its Voronoi cell is still closer to its previous position than any other node is, helping to roughly maintain the layout's shape. This method requires the least amount of extra space, but is much more destructive of the shape of the graph and the relative positions of the nodes.

In the implementation, node overlap is computed at the polygon level using a simplified version of the linear algorithm described in O'Rourke [33], preceded by a quick bounding box check. Non-polygonal nodes are approximated by polygons, typically using around 20 sides. In addition, the polygons are scaled up slightly to guarantee that on termination, there will be a clear positive distance between nodes. We use Fortune's $O(n\log n)$ sweepline algorithm [12] to compute Voronoi diagrams.

Several characteristics of this heuristic deserve further investigation.

- Counterintuitively, it runs faster while producing comparable layouts when all the nodes are moved on every iteration, instead of only moving overlapping nodes.
- It ignores edges. Better layouts could probably be made by incorporating edge information, for example, as part of the original layout.
- Unnecessary distortion of the graph's original shape occurs because the procedure expands the graph to fill the bounding rectangle. It would be interesting to try different bounding polygons, such as convex hulls or star-shaped outlines.

Once node overlaps are removed, the user has the option of avoiding node-edge overlaps by invoking a spline path planner module, as described in Section 3.4; edge-edge intersections are not considered.

3.3 Radial layout

Graphviz also provides an implementation (Algorithm 3) of a radial layout based on an algorithm of Eades [11] previously adapted by Wills [37]. Given a

Algorithm 3: Radial layout

Input : Graph G, vertex $c \in V$, $S > 0$
Output: Radial layout of G with c in the center

Construct rooted spanning tree T with c as root
foreach $v \in V$ **do**
  Let $size_v$ be the number of leaves in subtree of T rooted at v
  Let $parent_v$ be the parent of v in T
  Let $dist_v$ be the path distance of v from c
end
$angle_c = 2\pi$
foreach $v \in V$ **do**
  $p = parent_v$
  $angle_v = (angle_p \cdot size_v)/size_p$
end
$\theta_c = 0$
foreach $v \in V$ **do**
  if $v == c$ **then**
    $\Theta = 0$
  else
    $\Theta = \theta_v - angle_v/2$
  end
  foreach *child* w *of* $v \in T$ **do**
    $\theta_w = \Theta + angle_w/2$
    $\Theta = \Theta + angle_w$
  end
end
foreach $v \in V$ **do**
  $H = S \cdot dist_v$
  $x_v = Hcos(\theta_v)$
  $y_v = Hsin(\theta_v)$
end

center node c, the spanning tree is constructed such that, for each node v, the path from v to c in the tree is a shortest path in G. This algorithm is extremely fast, and works well with large graphs, typically representing nodes as points

and encoding additional attributes by color. A drawback of this layout is that the effectiveness of the diagram is very dependent on the choice of the center node. If the user does not supply a center, the implementation picks a "most central" node, i.e., one whose shortest distance to a leaf node is maximal. If there are no leaf nodes, a node is picked at random. This procedure is not unreasonable, since these types of radial layouts, especially if the graph is large, are only effective if the input graph is tree-like with low edge density.

3.4 Utility algorithms

Graphviz implements several general-purpose geometric algorithms to handle tasks which arise in almost all layouts. We discuss two of these here.

Spline path planner. Both for reasons of aesthetics and clarity, Graphviz gives the user the ability to draw edges as smooth curves. To accomplish this, we have implemented a general-purpose spline path planner, which will construct a spline curve between two points while avoiding nodes.

The spline path planner is a two-phase heuristic, as given in Algorithm 4. The procedure starts with the desired endpoints of the edge, typically clipped to the boundary of the node shapes, and a polygonal region. The polygon need not be simply connected. The polygon will usually contain at least the nodes of the graph, but may be modified further to additionally constrain the path. The first phase determines a shortest path connecting the two endpoints in the visibility graph of the vertices of the polygon. With a running time of $O(n^3)$ for the visibility graph computation, where n is the number of polygon vertices, this is only practical on modest-sized graphs.

Algorithm 4: Path planning heuristic

Input : Polygonal region P, points s and t in P
Output: B-spline C connecting s and t with C inside P

```
Construct visibility graph VG induced by s, t and the vertices in P
Find a shortest path L connecting s and t in VG
Construct Bezier curve C connecting s and t and fitting L
if C ∩ ext(P) ≠ ∅ then
    Adjust initial and terminal tangents
    if C ∩ ext(P) ≠ ∅ then
        Pick v on L furthest from C
        Replace L by the two paths [s, v] and [v, t] and recurse
    end
end
```

The second phase takes this piecewise-linear shortest path L connecting the two given endpoints and fits a candidate curve C to the path using the

algorithm of Schneider [18]. If the resulting curve remains on or within P, we are done. If not, we perform small adjustments to the tangents at the endpoints, bowing or flattening the curve, and stop if any of these variants work. If none do, we continue by recursion. This is done by picking a point v on L furthest from C, dividing the path at this point into two paths L_1 and L_2, and solving the each path separately. We maintain tangent information at v in order to combine the two solutions into a single B-spline that is C^1-continuous.

Note this technique offers no guarantee that the resulting spline topologically matches the original path, or that any of the path points except the endpoints are included. Our rationale is that the topological equivalence condition is difficult to check and is not usually a problem in practice, and forcing intersection with the path points often causes unwanted inflections in the curve.

In certain situations, the time for the first phase can be significantly improved. If we can guarantee that the polygon is simply connected, we can construct a shortest path using the "funnel" algorithm of Hershberger and Snoeyink [20] in time $O(nlogn)$. This is usually the case in hierarchical layouts, where it is easy to specify the polygonal region as the union of a set of contiguous, isotropic rectangles.

The spline router fits only one edge at a time; unwanted edge-edge intersections or tangencies can arise in routing multiple edges serially, whether between the same or different endpoints. To obtain effective global routing, the calling code needs to tailor the set of obstacles for each edge. Even when this is done, the splines created will typically be affected by the order in which they are created. One reasonable convention is to construct the shorter edges first.

Packing disconnected graphs. Most graph layout algorithms assume that the graph is connected. Given a disconnected graph, one can either apply the basic algorithm to each connected component and then arrange the components, or make the graph connected. The first approach is used by the Graphviz hierarchical layout. It aligns the highest rank of each component on a single line, as long as no additional rank constraints have been specified. By default, our implementation of Kamada-Kawai takes the second approach, setting the desired distance between every pair of nodes in separate components to $L/(|E| + \sqrt{|V|} + 1)$, where L is the sum of the lengths of all edges. This is large enough to guarantee that disjoint components do not overlap. Neither of these particular solutions is ideal, the former producing poor aspect ratios when there are many components, while the latter, though producing an attractive layout of central large galaxies surrounded by a ring of smaller systems, wastes a great deal of space.

To avoid these situations as well as to provide a general-purpose technique for combining disconnected graphs, Graphviz has a graph packing library

based on the polyomino packing algorithm [13] of Freivalds et al. There is an additional benefit of using this approach with Kamada-Kawai, as its basic algorithm has $O(n^2)$ complexity. If the graph is of medium size, say around 1000 vertices, but with many small components, applying it to each component and then packing the layouts together can improve the layout time by several orders of magnitude.

3.5 Dynamic *k*-layered drawing

While batch layout suffices in many applications, there are others when graphs are intrinsically dynamic and layouts need to be changed incrementally. For example, in an interactive graph editor, users edit graphs with an expectation of layout stability, or perhaps manually adjust the placement of some graph objects while others are being managed automatically. This becomes critical in the context of browsing huge graphs, where the user will need to view adjustable subgraphs or abstractions of a graph through a series of incrementally generated views.

It is possible that, when a small change is made to a graph, a static layout could be replaced, perhaps with the aid of some animation, by a new layout, provided the algorithm is stable under small changes. Typically, though, static algorithms are designed to perform global optimizations, and small changes in the graph can produce dramatic changes in the layout. The central problem, then, is how to make dynamic graph layouts which present readable, aesthetically-pleasing layouts, at each stage close to what one would obtain from a static algorithm, while highlighting changes and preserving a human's sense of context.

Dynagraph serves as the incremental version of Graphviz. The algorithms maintain a model graph with layout information, and accept a sequence of insert, modify or delete subgraph requests, with the subgraphs specifying the nodes and edges involved. The algorithms then adjust the model graph to reflect the layout changes, and generate a sequence of corresponding change messages by which the application can alter its version of the graph.

To give a flavor of the algorithms in Dynagraph, we focus on its incremental version [32] of a Sugiyama-type layout. This is a good example, since the standard static layout consists of 3 phases, with each phase performing a global optimization and with the output of one phase highly dependent on its input from the previous phase. After preprocessing the change sequence, handling requests which can be folded or canceled, the incremental layout still relies on the 3 standard passes.

It first handles ranks, reassigning levels to the nodes to maintain the hierarchy, preserve stability, and minimize total edge length, prioritized in that order. It employs the same network simplex algorithm used in the static case (Section 3.1), but with additional constraints to enforce stability. The added variables and constraints penalize level assignments by their variance from some given assignment, usually the previous layout. Adjusting the penalty

edge weights changes the tradeoff between minimizing edge length and maintaining geometric stability. Note also that there is no attempt to check for cycles in an earlier pass; when an edge causing a cycle is encountered during ranking, the edge will be reversed.

After all nodes have been assigned a new rank, the algorithm updates the configuration, converting long edges into chains of nodes as usual. It first moves the pre-existing nodes or node chains to match the new ranks just assigned. Then it moves edges by moving the chains to the new ranks, lengthening or clipping them as necessary.

At this point, the model graph has incorporated all of the requested changes. However, as might be expected, the edges in the layout probably have more crossings and bends than necessary. The next step is to reduce edge crossings. To do this, the algorithm identifies the neighborhoods of nodes and edges where insertion or modification have taken place, and applies a variant of the static crossing reduction heuristic to them. Aping the static case, the heuristic relies on multiple passes up and down the neighborhood, applying the median heuristic (Section 4.2 of Chapter 2) and local transpositions.

As in the static algorithm, the final pass involves computing the horizontal coordinates of the nodes. It again follows the static algorithm used in Graphviz, encoding the coordinates in an integer program which is solved using network simplex. As with ranking, the static constraints are extended and the objective function is modified to penalize changes from the current configuration.

Once all nodes are repositioned, the algorithm recomputes edge routes as necessary. Of course, new edges must always be routed, and existing edges are rerouted if an endpoint has moved, or the edge passes too near another node or edge. Edges are routed and drawn using the Graphviz path planner (Section 3.4).

4 Implementation

The design and implementation of Graphviz reflect the age of the software, with much of the core written in the early 1990's. Most of Graphviz is written in C, as it predates stable, efficient or portable implementations of C++ and Java[3]. The supporting libraries consist of some 45,000 lines of code, but two-thirds of that comes from our use of the GD graphics library [3]. The hierarchical layout requires about 6,000 lines; Kamada-Kawai, 3700; spring embedder for compound graphs, 2500; and radial layout, 400. The lefty graphical editor [26] is written in about 16,000 lines, with an additional 3000 lines of lefty script to tailor it for displaying and editing graphs.

The Graphviz design incorporates an interface level amenable to stream-processing filters for use with scripting languages. Though a library API as

[3] If the choice had to be made now, the same decision might be made for the same reasons.

well as interactive GUI interfaces are necessary and provided, we believe a well-designed scripting interface can greatly magnify the usefulness of software. A consideration of the many applications in which Graphviz tools have been used (cf. Section 2, Section 5 and [17]), and the simplicity of creating them, bear this out. Here, we mention a simple example. The hierarchical layout program draws disconnected graphs by placing the top rank of each component on the same rank. If there are a great many components, this produces a very wide but very shallow drawing. If this is unacceptable, a simple solution is to stream the graph into a tool that decomposes the graph into a stream of connected components, which in turn is fed into the layout program. This will layout each component, generating a stream of positioned graphs. This stream can then be fed into a packing filter, which combines each of the individual layouts into single layout with a much better aspect ratio. Finally, this graph can be piped into a tool which renders the drawing in the desired output format.

Another aspect that distinguishes the Graphviz software is its emphasis on providing the user with a rich collection of graphical primitives and output formats. Implementing various layout styles in order to view a graph's abstract features and topology is not enough. Graphviz was engineered to produce concrete pictures, in which the user has a wide choice in how semantic information and contextual attributes can be encoded. There are 24 basic node shapes, with most shapes having additional attributes for further customization. Nodes can also be drawn as records (see Fig. 3), basically rectangular arrays of text useful for representing data structures, or from user-supplied bitmaps or PostScript code. The user can chose from about 20 different arrowhead shapes, a variety of line styles for edges, most standard font formats such a Truetype and PostScript fonts, and the standard RGB and HSV color models. Graphviz also supports about two dozen output formats.

4.1 Architecture

Graphviz has a conventional layered architecture. At the center are a collection of libraries. The *libgraph* library provides the fundamental graph model, implementing graphs, nodes, edges and subgraphs, along with their related attributes and functions for file I/O. In turn, this library is built on Vo's *libcdt* [36] for the underlying set operations implemented as splay trees. Auxiliary libraries include the spline path planning library (Section 3.4); a version *libgd* of the GD library, which allows Graphviz to provide bitmap output in many standard formats as well use of the Freetype font engine; and a library for splitting graphs into connected components, and later combining the drawings of components into a single drawing.

At the next level, we have a core graph drawing library. This encapsulates the common parts of all layout algorithms, reading graph input, setting up the common data structures and attributes and, at the end, providing the drivers

for the two dozen or so output formats supported by Graphviz. Parallel with the core drawing library are the libraries coding each of the layout algorithms.

The next layer consists of the stand-alone programs. With the given libraries, these are basically just a main routine, which processes the command line flags and arguments, and then calls the appropriate library routines to read, layout and render a graph.

The top layer of interactive graph viewers and editors are built, in the main, from generic language and graphical interfaces [26,28], using the Graphviz layout programs as co-processes.

Dynagraph forms a parallel collection of layers for iterative graph layout algorithms. Most of Dynagraph is written in C++, and makes extensive use of C++ templates for code reuse. At the lowest level, Dynagraph creates a C++ API for *libgraph*. Higher layers define algorithms which produce standard layouts such as hierarchical and symmetric drawings, but rely on incremental techniques so that small changes to a graph produce small changes to the drawing. For Dynagraph, software to render graphs in a variety of concrete formats is unimportant. Rather, Dynagraph defines a graph editing protocol [32], which can be used by an application to feed incremental graph changes to the Dynagraph layout engines and, in return, receive descriptions of the incremental changes required in the drawing.

5 Examples

Fig. 4. Information visualization with graph drawings on a 10 megapixel display wall. The displays include maps of virtual private networks, a section of the public Internet, and software engineering diagrams. The visualizations are an important complement to conventional text and statistical displays for exploring large, semi-structured information sets

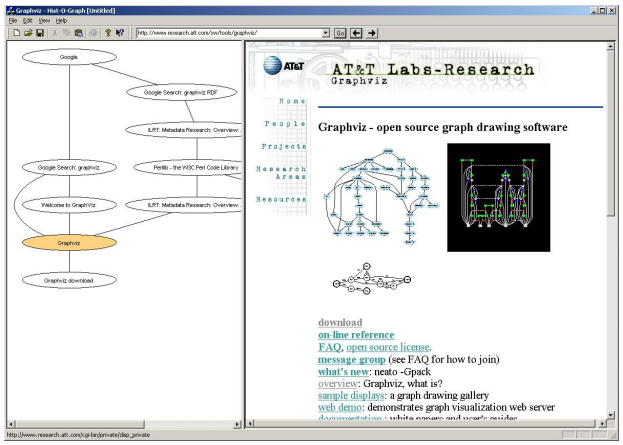

Fig. 5. Sample Histograph session. The right pane is a web browser; the left pane is a clickable history of the pages visited which is extended incrementally. The application illustrates integration of dynamic graph layout with other interactive tools

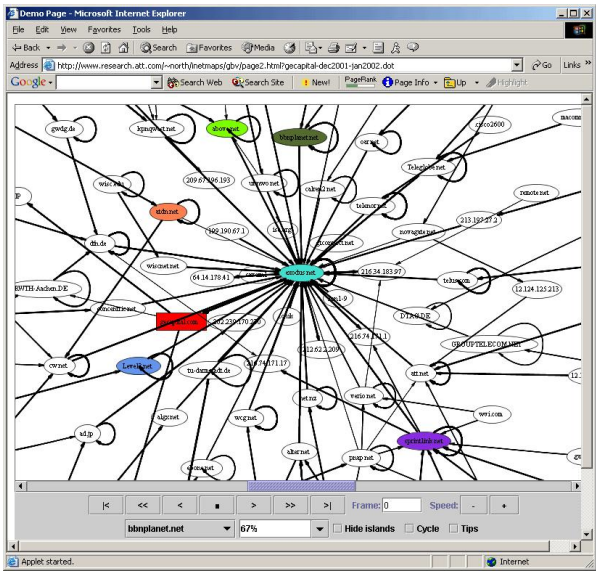

Fig. 6. Internet traceroute map viewer (courtesy of David Dobkin and John Mocenigo). The viewer plays an animation of routes from about 100 traceroute servers worldwide to a predefined list of target hosts. The routes are collected daily. The user interface is based on Grappa, the Java client from Graphviz, with added controls for animation playback and graph search

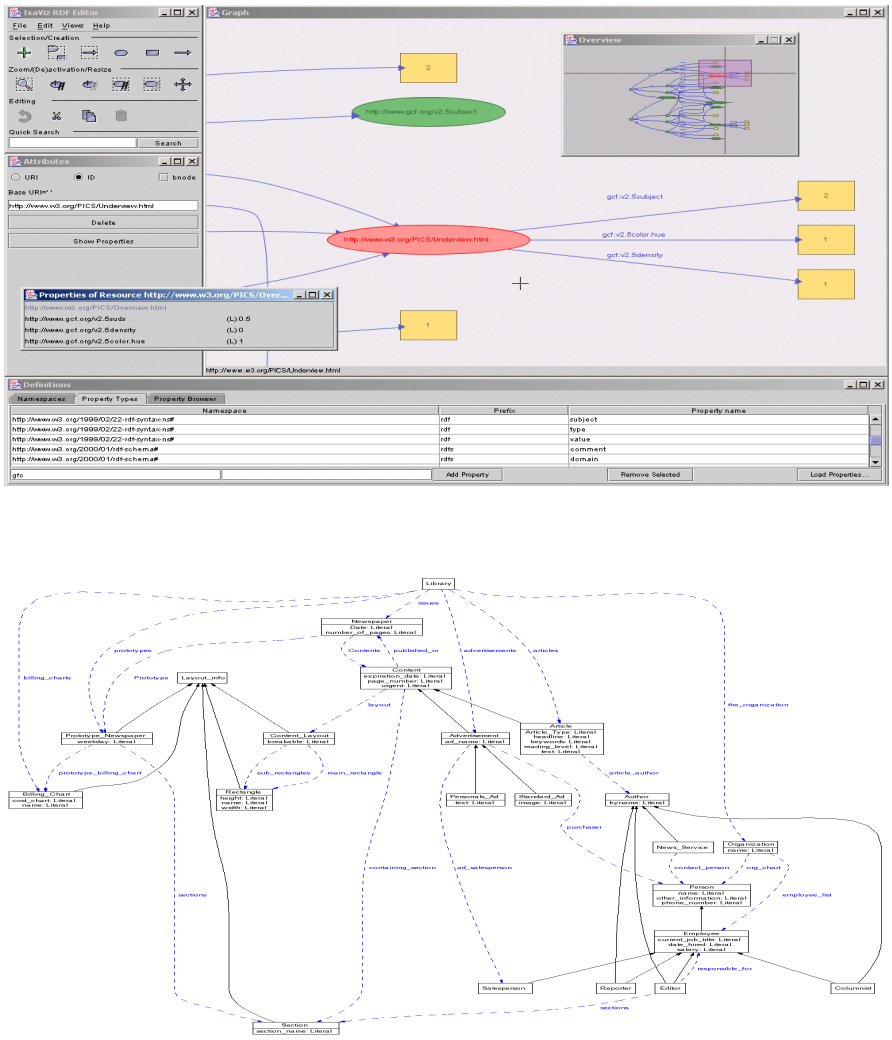

Fig. 7. Views from the IsaViz and RDFSViz RDF visualization tools (courtesy Emmanual Pietriga, W3C, and Michael Sintek, FRODO project at DFKI Kaiserslautern, respectively). IsaViz **(a)** is an RDF browsing and authoring tool which includes a 2.5D viewer based on the Xerox Visual Transformation Machine and is built on the Jena Semantic Web Toolkit from HP Labs and the Xerces XML parser from the Apache XML project. RDFSViz **(b)** is a schema ontology visualization tool built on the Java RDF API from Sergey Melnik at Stanford University and Xerces. It has become part of a related tool, OntoViz, from the Stanford Medical Informatics project Protege 2000, with more than 5000 registered users

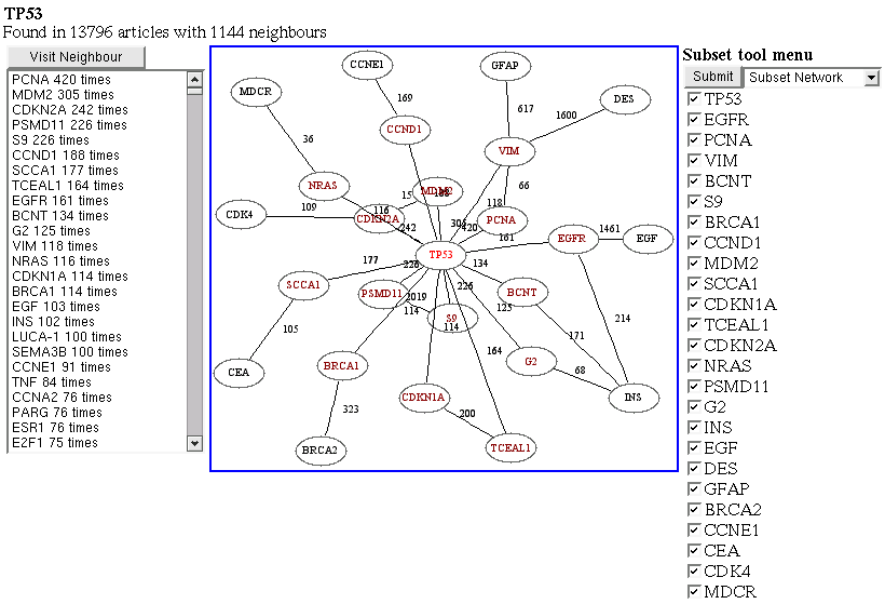

Fig. 8. PubGene provides web access to gene co-citation graphs for papers on rat, mouse and human genetics. Two genes mentioned in the same paper are connected by an edge. The project was created by a collaboration between the Department of Computer and Information Science and the Department of Physiology and Biomedical Engineering, Norwegian University of Science and Technology, Trondheim, Norway, and the Department of Tumor Biology, Institute for Cancer Research/Norwegian Radium Hospital, Oslo, Norway. It is now commercially supported by PubGene AS, Oslo, Norway

6 Software

The Graphviz software is freely available under an open source license. It is available at `www.graphviz.org` and at `www.research.att.com/sw/tools/graphviz`. In addition to software, the latter site also provides documentation, sample layouts and links to various sites describing libraries or packages incorporating uses of Graphviz.

References

1. Matthew P. Barnson. The Bugzilla guide, 2002. `www.bugzilla.org/docs/html/`.
2. Gregory W. Bond, Eric Cheung, K. Hal Purdy, J. Christopher Ramming, and Pamela Zave. An open architecture for next-generation telecommunication service. *ACM Transactions on Internet Technology*. submitted.
3. Thomas Boutell. GD graphics library, 2002. `www.boutell.com/gd/`.
4. Yih-Farn Chen, Emden R. Gansner, and Eleftherios Koutsofios. A C++ data model supporting reachability analysis and dead code detection. *IEEE Transactions on Software Engineering*, 24(9):682–693, 1998.
5. Yih-Farn Chen and Eleftherios Koutsofios. WebCiao: A website visualization and tracking system, 1997.
6. Victor Chvatal. *Linear Programming*. W. H. Freeman, New York, 1983.
7. Gnucleus: An Open Source Gnutella Client, 2002. `www.gnucleus.net`.
8. Jonathan Cohen. Drawing graphs to convey proximity: an incremental arrangement method. *ACM Transactions on Computer-Human Interaction*, 4(11):197–229, 1987.
9. William J. Cook, William H. Cunningham, William R. Pulleyblank, and Alexander Schrijver. *Combinatorial Optimization*. John Wiley and Sons, 1998.
10. Miguel de Icaza. The mono project: An overview, 2001. `developer.ximian.com/articles/whitepapers/mono/`.
11. Peter Eades. Drawing free trees. *Bulletin of the Institute for Combinatorics and its Applications*, 5:10–36, 1992.
12. Steven Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.
13. K. Freivalds, U. Dogrusoz, and P. Kikusts. Disconnected graph layout and the polyomino packing approach. In Petra Mutzel et al., editor, *Symposium on Graph Drawing GD'01*, volume 2265 of *Lecture Notes in Computer Science*, pages 378–391, 2002.
14. Thomas M.J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software – Practice and Experience*, 21(11):1129–1164, 1991.
15. Emden R. Gansner. The DOT language, 2002. `www.research.att.com/~erg/graphviz/info/lang.html/`.
16. Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, March 1993.

17. Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software – Practice and Experience*, 30:1203–1233, 2000.
18. Andrew S. Glassner, editor. *Graphics Gems*, pages 612–626. Academic Press, 1990. An algorithm for automatically fitting digitized curves.
19. David Harel and Yehuda Koren. Drawing graphs with non-uniform vertices. In *Proceedings of Advanced Visual Interfaces (AVI'02)*, pages 157–166. ACM Press, 2002.
20. John Hershberger and Jack Snoeyink. Computing minimum length paths of a given homotopy class. In *Proc. 2nd Workshop Algorithms Data Struct.*, volume 519 of *Lecture Notes in Computer Science*, pages 331–342. Springer-Verlag, 1991.
21. Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. `spinroot.com/spin/whatispin.html/`.
22. Bjorn Isaksson. DNS Bajaj, 2001. `www.foobar.tm/dns`.
23. Tor-Kristian Jenssen, Astrid Laegreid, Jan Komorowski, and Eivind Hovig. A literature network of human genes for high-throughput analysis of gene expression. *Nature Genetics*, 28:21–28, 2001. `www.pubgene.com`.
24. Paul Johnston. Syntacs translation toolkit, 2002. `inxar.org/syntacs/`.
25. Tomihisa Kamada and Satoru Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31:7–15, 1989.
26. Eleftherios Koutsofios and David Dobkin. LEFTY: A two-view editor for technical pictures. In *Graphics Interface '91*, pages 68–76, 1991.
27. Joseph Kruskal and Judith Seery. Designing network diagrams. In *Proc. First General Conference on Social Graphics*, pages 22–50, 1980.
28. Wenke Lee, Naser Barghouti, and John Mocenigo. Grappa: A graph package in Java. In *Symposium on Graph Drawing, GD '97*, pages 336–343, September 1997.
29. Kelly Lyons, Henk Meijer, and David Rappaport. Algorithms for cluster busting in anchored graph drawing. *Journal of Graph Algorithms and Applications*, 2(1):1–24, 1998.
30. Kim Marriott, Peter J. Stuckey, V. Tam, and Weiqing He. Removing node overlapping in graph layout using constrained optimization. *Constraints*, pages 1–31, in press.
31. Kazuo Misue, Peter Eades, Wai Lai, and Kozo Sugiyama. Layout adjustment and the mental map. *J. Visual Languages and Computing*, 6(2):183–210, 1995.
32. Stephen C. North and Gordon Woodhull. Online hierarchical graph drawing. In *Symposium on Graph Drawing, GD '01*, pages 232–246, September 2001.
33. Joseph O'Rourke. *Computational Geometry in C.* Cambridge University Press, Cambridge, 1994.
34. Douglas Thain. netmap, 2000. `www.cs.wisc.edu/~thain/projects/netmap`.
35. Dimitri van Heesch. Doxygen, 2002. `//www.stack.nl/~dimitri/doxygen/`.
36. Kiem-Phong Vo. Libcdt: A general and efficient container data type library. In *Proceedings of Summer '97 Usenix Conference*, 1997.
37. Graham Wills. Nicheworks – interactive visualization of very large graphs. In Giuseppe DiBattista, editor, *Symposium on Graph Drawing GD'97*, volume 1353 of *Lecture Notes in Computer Science*, pages 403–414, 1997.
38. Limsoon Wong. A protein interaction extraction system. In *Pacific Symposium on Biocomputing 6*, pages 520–531, 2001.