2006

Interactive, tree-based graph visualization

Andrew Pavlo

Interactive, Tree-Based Graph Visualization

Andy Pavlo

March 17, 2006

Abstract

We introduce an interactive graph visualization scheme that allows users to explore graphs by viewing them as a sequence of spanning trees, rather than the entire graph all at once. The user determines which spanning trees are displayed by selecting a vertex from the graph to be the root. Our main contributions are a graph drawing algorithm that generates meaningful representations of graphs using extracted spanning trees, and a graph animation algorithm for creating smooth, continuous transitions between graph drawings. We conduct experiments to measure how well our algorithms visualize graphs and compare them to another visualization scheme.

Contents

1 Introduction

Many real world problems can be modeled by graphs. By using graph visualization [34, 39, 71] to represent entities and relationships in these problems, one can see patterns that may be hard to detect nonvisually [45, 47, 67]. This is because humans are highly attuned to extracting information from a visual representation of a graph [5]. Graph visualization is important in many research areas of computer science, including software engineering [13, 14, 60], database design [1, 12], and networking [2, 3, 9, 26].

A major concern in graph visualization is how to create meaningful and useful representations of graphs automatically. Prior usability studies have shown that minimizing the number edge crossings in a graph enhances its readability [39, 58, 70]. However, it is usually impossible to draw complex graphs without crossings.

Our research is thus concerned with how to address the problems of edge crossings in graph drawings. We look to trees as an inspiration: (1) tree structures occur in nature, (2) trees can always be drawn without edge crossings, and (3) trees explicitly convey connectivity and distance relationships between elements of a graph. We developed a drawing algorithm that creates radial vertex-centric drawings of graphs using spanning trees extracted by breadth-first search. By reducing a graph to a spanning tree view, users can perceive a coherent segment of the graph's structure without being overwhelmed. In each drawing, the root of the tree is placed at the center of the viewing plane surrounded by its children vertices in a circle. The subtree rooted at each child in the graph is drawn on a series of overlapping circles. The graph's layout in the drawing has a recursive, self-similar structure found in natural trees.

Two problems arise, however, when using a spanning-tree-based drawing to visualize a graph. First, the view of the tree is heavily dependent on the root chosen; it is difficult to define the properties of a root vertex that will result in an informative spanning tree drawing. Second, removing edges from a graph to extract a spanning tree removes information.

To rectify these problems, we created an interactive visualization system on top of our drawing algorithm that allows users to explore a graph by viewing a sequential number of spanning tree drawings of the graph rooted at different vertices. An interactive environment helps mitigate the loss of information by reintroducing it over time via user interaction, graph animation, and multiple graph drawings. Users can select any vertex to become the root of a breadth-first spanning tree used in a vertex-centric drawing. We also developed an animation algorithm to generate smooth transitions as users select new drawings for viewing. Allowing users to view a graph in many different spanning tree layouts can facilitate the discovery of what makes one tree layout better than another.

A large collection of work exists on how to create drawings for graphs [8, 7, 27, 46, 52], including using three-dimensional images [40, 43, 44, 71]. Our system only creates two-dimensional drawings; creating a three-dimensional image does not alleviate the problems of edge crossings since the drawings are always projected down into two-dimensions. Previous research also uses graph drawings in an interactive environment to enhance users' understanding of a graph [33, 35, 41, 50, 54, 72, 73]. Interactive systems often use graph animation to transition the visualization display to a new view that reflects changes made to the graph [10, 17, 21, 25, 29, 30, 37, 38, 53, 73]. Usability studies have also been conducted to help understand how humans perceive graphs [39, 49, 51, 58, 70].

Radial layout graph drawing algorithms, such as ours, are a well known method for visualizing rooted trees [20, 41, 48, 66, 72, 73]. These approaches often construct drawings using concentric circles emanating from a focal point in the graph [20, 73]. Our drawing algorithm also uses circles to organize vertices but each circle is centered at the root of different subtrees in the graph and not the root of the entire tree. Other radial layout drawing algorithms use a similar approach to ours but instead position vertices on the entire length of circles and place subtree circles within one another [48, 72]. In our drawings, only the root's circle is used entirely for the placement of its children and no circle is completely contained inside another.

Previous research on interactive systems visualize graphs in spanning-tree-based layouts, but many of these systems require users to explicitly provide a single tree decomposition of a graph as its input and do not let users return to a view of the original graph [40, 46, 50]. In contrast, our system does not limit users to a single spanning tree drawing, and lets users switch back to view the full graph drawing at anytime.

To validate our drawing and animation algorithms, we conduct four experiments on random graphs that measure two aspects of graph visualizations. We measure the number of edge crossings that occur during three transition scenarios and the edge lengths of sets of sibling vertices in static drawings. For comparison, we implement the radial graph drawing and animation algorithms from Yee, Fisher, Dhamija, and Hearst's Gnutellavision graph visualization system and run the experiments under the same conditions [73]. One nice property of our graph drawings is that sibling vertices are always equidistant to their parent. In our experimental analysis, we evaluate the degree to which Yee et al.'s algorithms fail to create drawings with this property. Our data shows that our algorithms are able to transition between multiple spanning tree drawings of a graph with fewer crossings.

Our system can also transition between multiple drawings of the same tree with no crossings but Yee et al.'s system cannot.

This thesis is organized in the following manner. In Section 2 we provide an overview of the key graph theory concepts used in our work. In Section 3 we discuss graph drawings and graph drawing algorithms. In Section 4 we discuss graph animation and graph animation algorithms. Section 5 contains a brief overview of user interaction facilitates in graph visualization systems. In Section 6 we discuss the specifics of our graph visualization system including our context-free radial graph drawing and graph animation algorithms. Lastly, in Section 7 we present our results from the experimental analysis of our work.

2 Graph Theory

The definitions that follow are adopted from Diestel [18].

A *graph* G is a pair (V, E) where V is a set called the *vertices* of G and E is a set of two-element subsets of V called the *edges* of G. For a graph G, the vertex set of G is sometimes denoted as $V(G)$ and its edge set as $E(G)$.

For an edge $\{u, v\} \in E$, the two vertices u and v are the *endpoints* of $\{u, v\}$, u and v are said to be *adjacent* to each other, and u (or v) and $\{u, v\}$ are said to be *incident* to each other. The *degree* of a vertex u is the number of edges incident to u. The *order* of a graph is the total number of vertices within the graph.

For a graph G, a graph G' is a *subgraph* of G if $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. A graph G is *complete* if an edge exists in G for every unique pair of vertices of G.

A *path* is a non-empty graph $P = (V, E)$ where $V = \{v_0, v_1, v_2, \ldots, v_k\}$ and $E = \{\{v_0, v_1\}, \{v_1, v_2\}, \ldots, \{v_{k-1}, v_k\}\}$. For a path P where $k \geq 3$, P

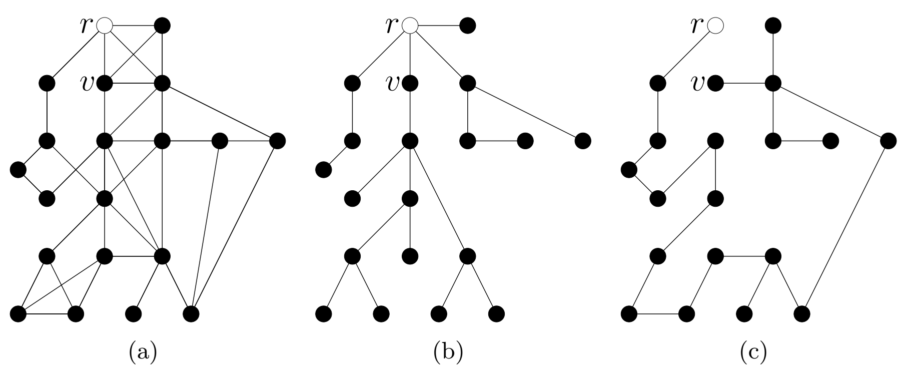


Figure 1: For a given graph G in Figure 1(a), Figure 1(b) contains a breadth-first spanning tree rooted at r extracted from G, and Figure 1(c) contains a depth-first spanning tree rooted at r extracted from G.

is a *cycle* if the edge $\{v_k, v_0\}$ is added to $V(P)$. A graph that contains no cycles as subgraphs is *acyclic*. A graph that has every unique pair of vertices in some path in the graph is *connected*.

A *tree* is an acyclic, connected graph. Every tree with n vertices has exactly $n-1$ edges. The vertices of a tree with a degree 1 are called its *leaves*. A *rooted tree* is a tree where one vertex is designated as the *root* of the graph. The *depth* of a vertex v in a tree rooted at r is the number of edges in the path from r to v. The *height* of a rooted tree is the number of edges in the longest path from the root to some leaf vertex in the tree. A vertex u in a rooted tree is the *parent* of vertex v if and only if u is adjacent to v and the depth of u in the tree is exactly one less than the depth of v. If u is the parent of v, then v is a *child* of u and its *siblings* are the set of all other children vertices of u. This relationship between a parent vertex and its children is one-to-many: a vertex can only have one parent but it can have many children.

A *spanning tree* of a graph G is a tree where its vertices are exactly all the vertices of G and its edges are a subset of the edges of G. Every connected graph has at least one spanning tree. Spanning trees are derived from graphs using traversal algorithms, such as depth-first or breadth-first search. The breadth-first spanning tree in Figure 1(b) and the depth-first spanning tree in Figure 1(c) illustrate how different traversal algorithms extract different trees for the same graph shown in Figure 1(a). Breadth-first spanning trees are particularly useful in graph theory and computer science because they preserve the distance from the root vertex to any other vertex: the shortest path distance from vertex v to the root r in Figure 1(b) is the same as it is in the original graph in Figure 1(a), however, v's shortest path distance from v to r is much greater in Figure 1(c) than in Figure 1(a) and Figure 1(b).

A graph drawing represents pictorially the structure of a graph by placing vertices at points on a plane and representing edges with curves between these points [18]. More formally, a drawing $\Gamma : V(G) \cup E(G) \rightarrow \mathbb{R}^2 \cup 2^{\mathbb{R}^2}$ for a graph G is a function that maps each vertex $v \in V(G)$ to a point $\Gamma(v)$ and each edge $\{u, v\} \in E(G)$, where $u, v \in V(G)$, to a open Jordan curve $\Gamma(\{u, v\})$ with endpoints $\Gamma(u)$ and $\Gamma(v)$ [8].

Two graphs $G = (V, E)$ and $G' = (V', E')$ are said to be *isomorphic* if there exists a bijection $\varphi : V \rightarrow V'$ such that for all $u, v \in V$, $\{u, v\} \in E \Leftrightarrow \{\varphi(u), \varphi(v)\} \in E'$. That is, G and G' are isomorphic if they contain the same number of vertices connected by edges in the same way.

3 Graph Drawings

The art and science of graph drawing is matching problem domains with algorithms and heuristics for producing visual representations of graphs.

In Section 3.1, we discuss three types of graph drawing parameters that describe how a graph is drawn. In Section 3.2, we examine three examples of layout algorithms using different models to calculate the positions of vertices and edges in a graph drawing.

3.1 Graph Drawing Parameters

Graph drawing parameters describe the visual characteristics of a graph drawing. A graph drawing algorithm can use these parameters to produce drawings that help users identify relevant attributes of a graph and facilitate quick recognition of its important properties [8]. Graph drawing parameters can accommodate the differences in spatial memory, reasoning abilities, and visual predilections that users may have [63]. This may limit the scope of a graph drawing system, and as such, many graph drawing parameters can only be used for specific kinds of graphs or produce better results for graphs of a certain type [8].

We now consider Battista, Eades, Tamassia, and Tollis's division of graph drawing parameters into three categories [8]: (1) general conventions for the geometric representation of a graph (Section 3.1.1), (2) layout aesthetics for producing a readable drawing (Section 3.1.2), and (3) constraints that subgraphs in drawing may be required to satisfy (Section 3.1.3).

3.1.1 Drawing Conventions

Drawing conventions are application-dependant criteria that a graph drawing must satisfy. When defined, conventions specify how vertices and edges are represented in a graph drawing. Below we include a list of drawing conventions from Battista et al. [8]. Figure 2 illustrates the differences of three drawings of the same graph using different drawing conventions.

Straight Line: Edges must be drawn as a single line segment without any bends.

Polyline: Each edge must be drawn as a chain of straight line segments.

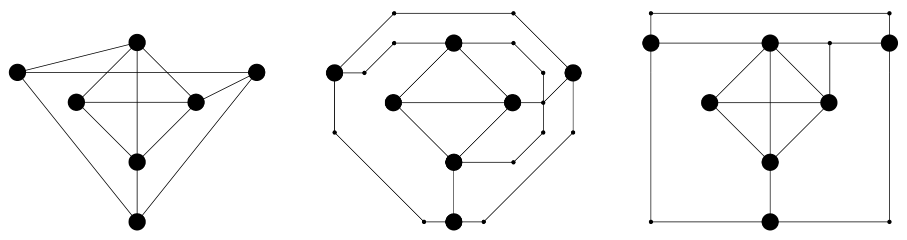


Figure 2: Three drawings of the same graph using different drawing conventions. The small circles at edge bends in Figure 2(b) and Figure 2(c) are added for emphasis.

Grid Drawing: Vertices must be positioned only at integer Cartesian coordinates on the drawing plane.

Planar Drawing: No two edges are allowed to intersect or overlap in the drawing.

Orthogonal Drawing: The graph must be drawn as a polyline grid drawing where the edges are only allowed to bend at right angles and are comprised of alternating horizontal and vertical segments.

3.1.2 Aesthetics

Aesthetics are measurable properties used to evaluate the quality of a graph drawing [6, 64]. Research suggests that if a drawing algorithm strongly adheres to one or more carefully chosen aesthetic goals, it can create more meaningful drawings [39, 58, 70]. Included below is Battista et al.'s list of common graph drawing aesthetics [8]. When developing a graph drawing algorithm, one seeks to optimize the presence of a particular aesthetic by either minimizing or maximizing the measurable quality of the aesthetic. Figure 3 shows three drawings of the same graph that emphasize different aesthetics.

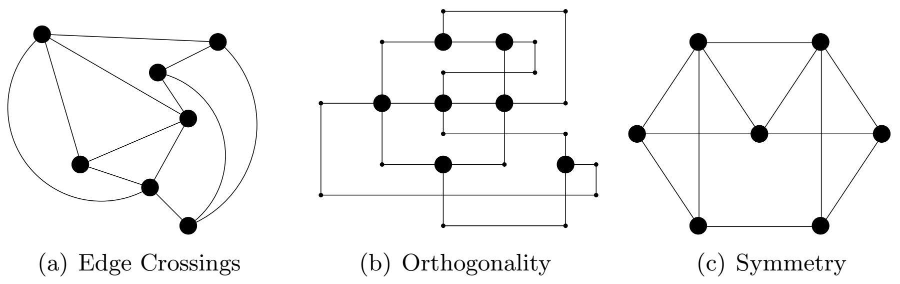

(a) Edge Crossings (b) Orthogonality (c) Symmetry

Figure 3: Three drawings of the same graph that adhere to different aesthetic goals. The drawing in Figure 3(a) is constructed to minimize the number of edge crossings. The drawing in Figure 3(b) is constructed to maximize the orthogonal properties of the graph. The drawing in Figure 3(c) is constructed to maximize the symmetrical nature of the graph.

Planarity / Edge Crossings: The number of edge crossings in the drawing.

Area: The total area of a drawing.

Edge Length: Measuring the variance, the maximum length, or the sum total of edge lengths for all vertices or sets of vertices in a drawing.

Edge Bends: The number of separate line segments in a polyline edge.

Angular Resolution: Measuring the size of the angle between edges incident to the same vertex.

Symmetry: Measuring the symmetrical properties of a graph drawing.

Orthogonality: Measuring the orthogonal nature of the graph drawing.

It is not always possible to optimize certain aesthetics within a graph drawing. For instance, not every graph admits a planar drawing and when no planar drawings exist it can be difficult to even find layouts that minimize edge crossings. One might attempt to turn a graph into a planar graph by finding the minimum number of edges that would need to be removed from a graph in order to make it planar. However, the problem of determining for a given number k and a graph G whether it is possible to make G planar by removing fewer than k of its edges is NP-hard [31].

Furthermore, as hard as it can be to optimize one aesthetic in a graph drawing, it is often more difficult or impossible to simultaneously optimize two or more [8]. A graph drawing algorithm must establish priorities for the aesthetics that it uses because aesthetics may conflict. For instance, although the drawings in Figure 3 depict the same graph, the drawing in Figure 3(b) contains edge crossings but the drawing in Figure 3(a) does not. It is a NP-hard problem in a straight-line graph drawing on an orthogonal grid to minimize crossings while trying to minimize the total number of edge bends [32]. An algorithm's preference of one aesthetic over others will influence the visual properties of the drawings it generates.

Purchase conducted studies on the effectiveness of various graph drawing aesthetics [58]. Her experiments test which aesthetics best predicts a user's ability to answer questions about a graph. For each aesthetic, two graph drawings were created; one having a strong presence of the aesthetic, and the other a weak presence. Users were asked simple questions about each drawing, such as what is the shortest path between two vertices. The reaction times and error rates for the questions determine how well a drawing aesthetic improves a user's ability to discern information from a graph.

Purchase concluded that the most significant factor for improving both reaction times and error rates for graph drawings is minimizing the number of edge crossings [58]. Additionally, minimizing the number of edge bends substantially improved error rates and maximizing perceptual symmetry slightly improved reaction times. Maximizing the orthogonal structure of the drawing and maximizing the angles between edges for adjacent vertices did not appear to have a significant effect on a user's performance.

Ware, Purchase, Colpoys, and McGill extended Purchase's work and studied the impact of multiple aesthetics within a single graph drawing instead of testing drawings representing the extremes of an aesthetics as Purchase did [70]. Ware et al.'s also found that minimizing the number of edge crossings in a graph drawing has the greatest positive impact on users. A study by Huang and Eades used an eye tracking system to observe users' eye movement patterns when viewing a graph drawing [39]. Like Ware et al., their results show that minimizing the number of crossings improves a user's ability to make judgments about a graph.

3.1.3 Drawing Constraints

Unlike drawing conventions and aesthetics, which are rules and criteria applied to the entire graph drawing, graphical *drawing constraints* are rules that refer to specific subgraphs and subdrawings [8]. Constraints often specify the position of elements based on semantic information about the graph. For instance, a constraint might specify that the most important vertex of a graph is positioned at a particular location in the drawing. The list below from Battista et al. contains commonly used constraints for drawing algorithms [8]. Figure 4 shows three drawings of the same graph conforming to different drawing constraints.

Centered Placement: Place a specific vertex at the center of the drawing.

External Placement: Place specific vertices at the outer areas of the drawing.

Clustered: Position vertex subsets close together.

Uniform Edge Lengths: The edges of a subgraph are all equal in length.

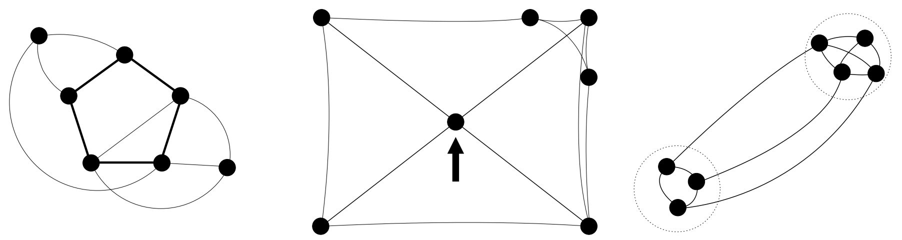

(a) Shape (b) Centered/External (c) Clustering

Figure 4: Three drawings of the same graph using different graph drawing constraints. In the drawing in Figure 4(a), a subgraph forms the shape of a pentagon. In the drawing in Figure 4(b), a single vertex is placed at the center of the drawing (indicated by the arrow), while all other vertices are placed on the external edges of the drawing plane. The drawing in Figure 4(c) contains two subgraph clusters of vertices positioned together (indicated by the circle outlines).

Shape: Draw a subgraph in a pre-defined shape. Vertices can be positioned on the outline of the shape or edges can be drawn to conform to the shape.

3.2 Drawing Algorithms

A drawing algorithm reads the description of a graph as its input, calculates the positions of the graph's elements on a viewing plane using graph drawing parameters as guidelines, and outputs a drawing [8]. There are many different classes of drawing algorithms [7]. We now examine a force-directed layout algorithm that creates drawings using a physical system model (Section 3.2.1), a hyperbolic layout algorithm that creates fisheye-distorted drawings (Section 3.2.2), and a radial layout algorithm that creates concentric-circle-based drawings (Section 3.2.3).

3.2.1 Force-Directed Layout

A force-directed drawing algorithm creates graph drawings by simulating repulsive and attractive forces [19]. Each vertex is assigned a repulsive force and an initial position, and each edge is given an attractive force between its endpoints. The resulting drawing represents a layout of the vertices and edges in a locally minimal energy state. Eades' original implementation of the force-directed algorithm models vertices as electrically charged particles that repel other vertices and edges as metal springs that pull adjacent vertices toward one another [19]. Each spring's pulling force becomes greater as adjacent vertices repel each other, causing the spring to stretch. These opposing forces cause non-adjacent vertices to move away from each other while adjacent vertices are held close together. The simulation eventually converges to an equilibrium state, where the pull at each vertex from the springs is equivalent to the repulsion of the vertices. Other, more complex force-directed models have been developed to use a more accurate representation of Hooke's law for the behavior of springs [42], and reduce energy factors through simulated annealing [16].

3.2.2 Hyperbolic Layout

Hyperbolic layout algorithms embed a graph's elements in hyperbolic space to produce drawings that have a distinct "fisheye" appearance. Like Euclidean space, hyperbolic space consists of points, lines, planes and surfaces. In hyperbolic space, however, there are many lines through a given point which do not intersect a given line [15, 56]. A hyperbolic graph layout algorithm creates a new drawing by first constructing a layout of the graph in hyperbolic space, and then projecting this layout on to the Euclidean plane or space [34]. As a result of this projection, graph elements are represented

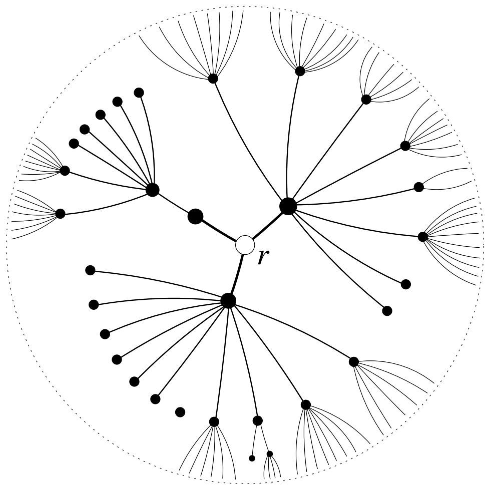


Figure 5: A hyperbolic layout graph drawing for a tree rooted at r (adapted from [46]). In hyperbolic drawings, a graph's elements are represented in space proportional to their distance to the root vertex; as vertices are positioned further away from the root they become vanishingly small.

in space proportional to their distance from origin of the drawing; as an element gets farther away from the origin, it is represented by smaller amounts of space. An example of a hyperbolic layout graph drawing is shown in Figure 5.

This distorted view of the graph is useful for viewing large hierarchies and data sets [34]. Lamping, Rao, and Pirolli's hyperbolic graph visualization system generates two-dimensional drawings for subsections of the World Wide Web [46]. Munzner expanded on their approach to project the graph onto a three-dimensional viewing plane [50]. More recently, the Walrus visualization tool by Hyun produces stunning hyperbolic visualizations for trees with over a million vertices [40].

3.2.3 Radial Layout

Radial layout graph drawing algorithms are a well known method for creating drawings of rooted trees [8, 20, 48, 66, 72, 73]. In the example in Figure

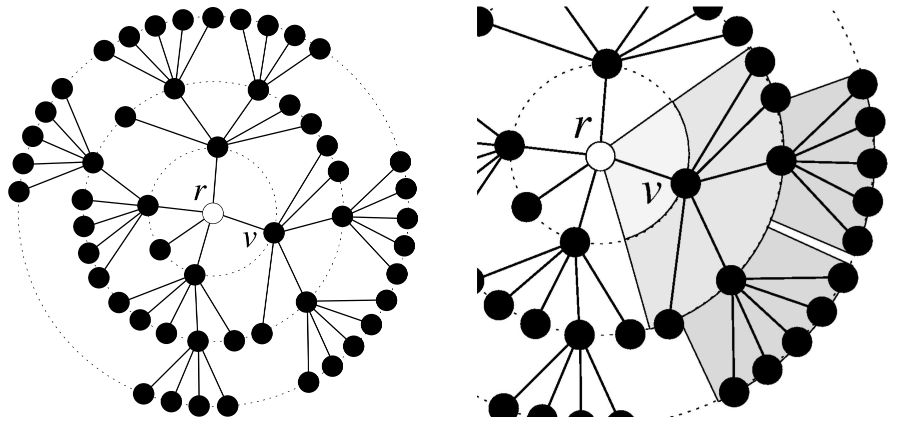


Figure 6: A radial layout graph drawing of a tree rooted at r (Figure 6(a)) and the division levels of v's annulus wedge (Figure 6(b)).

6(a), the root vertex of a tree is positioned at the center of the drawing with descendant vertices situated on concentric circles emanating from it. For a tree T with a height k, the layers of concentric circles are labeled as $C_1, C_2, \ldots, C_k$ and each vertex is placed on circle C_i, where i is the depth of the vertex in the rooted tree [20, 59]. Using a defined heuristic, a radial layout drawing algorithm allocates each vertex space in the drawing, known as its *annulus wedge*. This wedge confines the layout of a vertex's subtree to particular area in the drawing. A vertex's annulus wedge is divided among its descendants at subsequent levels in the subtree (Figure 6(b)).

For a tree T rooted at a vertex r, the *radial_positions* algorithm (page 18) outputs a new radial layout graph drawing Γ for T. The algorithm first places the root vertex at the center of the viewing plane and allocates it an annulus wedge of the entire drawing (360°). This space is divided among the root's descendants: the annulus wedge for a child vertex c of v is based on the number of leaf vertices in the subtree rooted at c proportional to the number of leaf vertices in the subtree rooted at v. Each vertex is placed at the center of its annulus wedge on a concentric circle corresponding to its

Algorithm 1 *radial_positions* $(T, v, \alpha, \beta, \Gamma)$

```
if ($v$ is the root of the tree $T$) then
  $\Gamma(v) \Leftarrow (0, 0)$
end if
$D \Leftarrow$ the depth of $v$ in $T$
$\Theta \Leftarrow \alpha$
//Calculate the radius for this concentric circle level
$R_D \Leftarrow R_0 + (\xi \cdot D)$
$\kappa \Leftarrow$ the number leaves in the subtree rooted at $v$
for all (children $c$ of $v$ in $T$) do
  $\lambda \Leftarrow$ the number leaves in the subtree rooted at $c$
  $\mu \Leftarrow \Theta + \left(\frac{\lambda}{\kappa} \cdot (\beta - \alpha)\right)$
  //Convert $c$'s polar coordinates to absolute Cartesian coordinates
  $\Gamma(c) \Leftarrow \left(R_D \cdot \cos\left(\frac{\Theta+\mu}{2}\right), R_D \cdot \sin\left(\frac{\Theta+\mu}{2}\right)\right)$
  if ((the number of children of $c$ in $T$) $> 0$) then
    $\Gamma \Leftarrow$ radial_positions $(T, c, \Theta, \mu, \Gamma)$
  end if
  $\Theta \Leftarrow \mu$
end for
output $\Gamma$
```

Figure 7: Given a rooted tree T, a vertex $v \in V(T)$, and the angles α and β that define v's annulus wedge, the algorithm calculates the position of every child vertex c of v in a new graph drawing Γ. R_0 is the user-defined radius of the innermost concentric circle. ξ is the user-defined delta angle constant for the drawing's concentric circles. The initial values for α and β for the root's annulus wedge are $0°$ and $360°$, respectively.

depth in the tree. The algorithm continues down each subtree until positions for all of the graph's vertices in the new drawing Γ are calculated.

Yee et al.'s Gnutellavision graph visualization tool uses the *radial_positions* algorithm to provide an overview of a peer-to-peer network [73]. In their system, users select a host from the network to be the root of a spanning-tree-based drawing. The concentric circles in their drawings represent the network distance from the root to all other hosts; Each vertex is placed on a circle corresponding to its shortest network distance to the root. A new view of the network is constructed every time the user selects a new focal point vertex.

Yee et al.'s system also employs particular graph drawing parameters

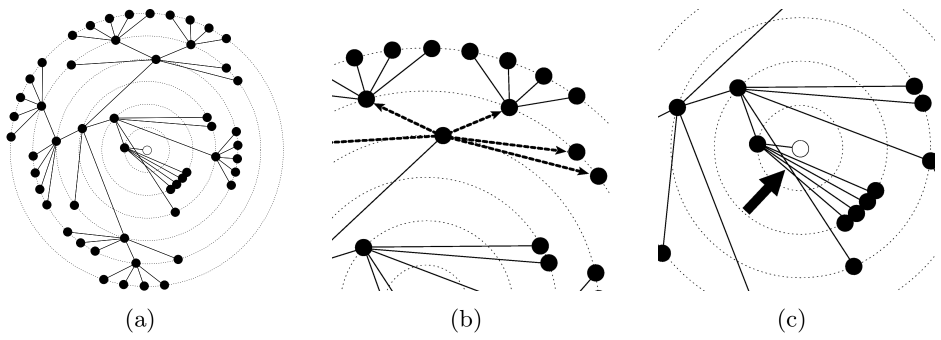


Figure 8: A radial layout graph drawing of the same tree from Figure 6, but with a different root vertex as the focal point. Figure 8(b) illustrates how sibling vertices have variable edge lengths to their parent, and Figure 8(c) highlights edge crossings in the drawing.

that help in creating sequences of drawings where a new drawing bears some resemblance to the previous [73]. For example, their algorithm mandates that the direction of the edge between a newly selected root vertex to its parent from the previous drawing is preserved. The new drawing also preserves the rotational ordering of children vertices around their parent from the previous drawing. Yee et al. believe that these features of their system help users relate one drawing of a graph to the next.

However, like with any graph drawing algorithm, the drawings generated by radial layout algorithms do have some drawbacks. Although the use of concentric circles does make it easier for users to ascertain the depth of a vertex in a tree, these circles confine vertices to positions that may not be optimal and can make it difficult for users to visually distinguish siblings from their parent. This is because sibling vertices may be spread out widely on their corresponding circle, and thus the lengths of the edges to their parent are dramatically different. For example, the drawings in Figure 8 depict the same tree from Figure 6 but based on a different root vertex. In Figure 8(b), the edges marked by arrows illustrate edges for sibling vertices that are different in length. Another problem is that radial layout drawings can still allow edge crossings, even if the graph is planar (Figure 8(c)). As the aesthetic usability studies have shown in Section 3.1.2, these flaws can degrade the readability of a graph.

We now examine how graph animation aids in the visualization of graphs.

4 Graph Animation

In a graph animation, each frame in the animation sequence is a single graph drawing that contains a subtle change from the previous drawing in the sequence. When the frames are viewed in succession at an adequate speed, the slight changes from one frame to the next is perceived as movement. The entire animation sequence creates the illusion of the graph moving from one layout to another.

Animation is an important feature in interactive graph visualization systems. Graph drawings in such system are susceptible to change by users and outside stimuli, and users need to be informed of these changes in a way that does not overwhelm them. Instead of instantaneously updating a drawing for every change that occurs, a well-designed animated transition can facilitate visual continuity over multiple drawings of a graph [17, 29, 30]. Graph animation should provide a smooth, continuous movement of a graph as a means for revealing structural differences between two drawings while preserving users' mental maps [29]. A transition that is both visually appealing and easy to follow helps users relate two separate graph drawings to one another [21, 29].

Our discussion of graph animation used in visualization systems is as follows. In Section 4.1, we discuss mental maps as they relate to graph animation. In Section 4.2, we discuss graph animation algorithms and present three particular algorithms as examples.

4.1 Viewer Mental Map

A *mental map* is a cognitive model of the spatial relationships between graph elements that users form when viewing a graph [4, 22, 49, 65]. The concept of a mental map is important to graph animation research because one seeks to create a transition between multiple drawings of a graph that preserves users' mental maps. A good graph animation should not hinder users from applying pre-existing knowledge about a graph to the new drawing. If it is too difficult to relate a new drawing to the previous, users may have to exert effort for each new drawing to regain familiarity with the graph [22].

Eades, Lai, Misue, and Sugiyama propose models for evaluating how well two graph drawings resemble each other as an indicator for whether users will be able to maintain mental map continuity when a visualization system switches from one drawing to another [22]. For graph animation, Friedrich and Eades propose models and metrics for how well an animation algorithm transitions a graph, however, they do not conduct any experiments [29]. To our knowledge, these metrics have not been used in human experiments.

4.2 Animation Algorithms

Given a graph and two drawings, which we call the initial and new layouts of the graph, a graph animation algorithm generates a series of drawings, called frames, of the graph that viewed sequentially, provides the illusion of a smooth, continuous transition from the initial layout to the new layout. As with graph drawing, animation algorithms are typically tailored to emphasize the characteristics of the graph that are most important to the intended audience [49].

One simple class of graph animation algorithms creates intermediate frames by linearly interpolating between the Cartesian coordinates of vertices in the initial and new layouts [28, 38]. Although such algorithms may work well on graphs with few elements, they are not practical for graphs where hundreds of vertices may need to move great distances. This kind of transition causes vertices to congregate together at particular locations in the drawing, creating large swarms. These swarms reduce a user's ability to maintain his or her mental map of the graph; it is difficult to determine which vertex came from which position when they are densely grouped together.

Alternative methods devised for generating animated graph transitions seek to avoid the drawbacks found in the previous linear interpolation example. We now describe three more sophisticated methods. First, in Section 4.2.1 we discuss an animation algorithm that creates rigid-body motion for graphs. Next, we discuss a derivative work in Section 4.2.2 that generates similar movements for individual subgraphs rather than a single movement for the entire graph. Finally, in Section 4.2.3 we discuss an animation algorithm for radial layout graph drawings.

4.2.1 Rigid-Body Animation

To alleviate vertex swarming problems that often occur in a transition derived from a linear translation of Cartesian coordinates, Friedrich and Eades' animation algorithm generates transition sequences that move a graph as a solid object instead of as a collection of individual vertices moving separately and erratically [29]. Such an approach is motivated by the belief that the human brain is predisposed to follow uniform movements of objects [51]. The transition is divided into separate rigid motion and force-directed layout stages. The graph first moves as a rigid object to position that is close to the new layout using only linear transformations [29]. The animation algorithm then completes the transition using a force-directed simulation where vertices are attracted to their destinations (instead of adjacent vertices).

Additionally, Friedrich and Eades use visual cues to help users anticipate the animation better. When vertices or edges are to be removed from the drawing during a transition, they gradually fade from view before the graph begins to move. Likewise, when new vertices or edges are being added to the drawing, they gradually fade into view only after the graph has reached its final destination. Friedrich and Eades remark that the removal and addition of graph elements before and after the transition reduces the amount of visual distractions to users [29].

4.2.2 Clustered Movement Animation

Friedrich and Eades' rigid-body animation is effective when a graph's vertices are uniformly distributed. The algorithm, however, performs poorly for graphs that contain subgraphs whose optimal movement conflict with the average optimal movement of the entire graph [30]. For example, even if only a small part of a graph is altered, rigid-body animation algorithm moves all vertices in the graph as a result. A derivative animation algorithm by Friedrich and Houle adopts the same rigid-body-motion techniques from Friedrich and Eades' previous work, but apply it individually to subgraphs [30]. Their algorithm uses clustering heuristics to identify subgraphs to share uniform motion. The resulting animation sequences transition a graph as separate components, each with its own unique movement. Users mentally group subgraphs as separate objects and can follow the movements more easily [51].

4.2.3 Radial Graph Animation

There are different animation methods for radial layout graph drawings [41, 73]. Although many of the same animation criteria still apply, radial layout transitions can have additional goals of maintaining vertex rotational ordering and preserving the boundaries of subtrees.

Given a rooted tree T, a vertex $v \in V(T)$, a time step s in the animation sequence, an initial drawing Γ_1 of T and a new drawing Γ_2 of T generated by the graph drawing algorithm in Section 3.2.3, the *radial_transition* algorithm (page 25) computes the positions of v's children at time step s and outputs an intermediate drawing frame Γ_s. The algorithm calculates the graph's movement by interpolating the polar coordinates of vertices' positions from Γ_1 to Γ_2. To generate all the intermediate frames for an animation, one would invoke this algorithm from time steps 1 to S, where S is the last frame in the animation sequence.

This animation algorithm creates two distinct types of motion for the vertices of T. The root of the tree moves in a straight-line path from its original position in Γ_1 to the center of the new drawing. All other non-root vertices move along radial paths to their new positions in Γ_2. The algorithm's interpolation of vertices' polar coordinates creates smooth transitions for radial graph drawings while avoiding occlusion problems that can occur using a linear translation of Cartesian coordinates [73]. This movement prevents vertices from amassing at the center of the drawing and occluding one another.

Yee et al.'s Gnutellavision application uses the *radial_transition* algorithm to generate animated transitions when users select a vertex to become the focal point of a new vertex-centric perspective of the Gnutella network [73]. Yee et al.'s animation uses a slow-in, slow-out timing method for the

Algorithm 2 *radial_transition* $(T, v, s, \Gamma_1, \Gamma_2, \Gamma_s)$

```
$\Delta \Leftarrow \frac{s}{S}$
**if** ($v$ is the root of the tree $T$) **then**
    //The root moves on a straight-line path to $(0, 0)$
    $(x, y) \Leftarrow \Gamma_1(v)$
    $\Gamma_s(v) \Leftarrow (x \cdot (1 - \Delta), y \cdot (1 - \Delta))$
**end if**
**for all** (children $c$ of $v$ in $T$) **do**
    $(\Theta_1, R_1) \Leftarrow ((\Gamma_1(c))_\theta, (\Gamma_1(c))_r)$
    $(\Theta_2, R_2) \Leftarrow ((\Gamma_2(c))_\theta, (\Gamma_2(c))_r)$
    $\theta \Leftarrow (\Theta_1 \cdot (1 - \Delta)) + (\Theta_2 \cdot \Delta)$
    $r \Leftarrow (R_1 \cdot (1 - \Delta)) + (R_2 \cdot \Delta)$
    //Convert $(\theta, r)$ to absolute Cartesian coordinates
    $\Gamma_s(c) \Leftarrow (r \cdot \cos(\theta), r \cdot \sin(\theta))$
    //Calculate animation frames for $c$'s children
    **if** ((the number of children of $c$ in $T$) $> 0$) **then**
        $\Gamma \Leftarrow$ *radial_transition* $(T, c, s, \Gamma_1, \Gamma_2, \Gamma_s)$
    **end if**
**end for**
**output** $\Gamma_s$
```

Figure 9: Given a rooted tree T, a vertex $v \in V(T)$, the initial drawing Γ_1 for T, a new drawing Γ_2 of T, and the current time step s in the animation, the algorithm calculates the positions of v's children at time step s and outputs an intermediate frame Γ_s. For any point $\vec{p}$ in a Cartesian coordinate system, let $(\vec{p}_\theta, \vec{p}_r)$ denote the polar coordinates of $\vec{p}$. Let Δ be the interpolation factor at time step s. Let S be the total number of frames in the animation sequence.

algorithm's interpolation factor (Section 6.2.3). Although only informal human experiments were conducted to validate this variable speed approach in radial graph animations, Yee et al. claim users' prefer exploring graphs with this feature.

We now discuss user interactivity as it applies to graph visualization systems.

5 User Interaction

Interactive visualization systems can often help users explore a graph more easily than a single, static drawing. Herman, Melançon, and Marshall's survey of graph visualization discusses some of the more prevalent user interaction techniques and facilities in graph visualization systems [34]. We would now like to highlight three important concepts from their work.

5.1 Zooming and Panning

Herman et al. first discuss two user interaction capabilities found in many visualization systems: *zooming* and *panning*. Visualization zooming can be either geometric or semantic. Geometric zooming scales a visualization system's viewing plane to create either coarse overviews or detailed perspectives of graphs. Semantic zooming alters the information content of a graph's elements according to some heuristic. Panning is simply a translation of the viewing plane for a graph drawing.

5.2 Focus+Context

The *focus+context* graph visualization scheme creates drawings where an area of interest in a graph is enlarged while other portions are shown with successively less detail [34]. This enlargement can either be displaying the focal point in greater detail and/or representing it with more geometric space in the drawing [41]. Users interact with the visualization system by changing the parameters that govern the focus+context distortion.

Herman et al. splits focus+context graph visualization implementations into two categories: (1) the distortion is applied to a graph drawing after it is generated, and (2) the distortion is integrated into the graph drawing algorithm [34]. An example of applying the distortion after the drawing is generated is the popular fisheye distortion technique [61, 62]. This approach imitates a wide-angle lens to enlarge the area surrounding the focal point in a display and shows peripheral areas of the graph with decreasing detail. The drawback to this implementation, according to Herman et al., is that because the distortion is applied after the drawing is generated, aesthetic adherence may be degraded. Other visualization paradigms implement the focus+context distortion directly in the graph drawing algorithm, such as the hyperbolic layout algorithm in Section 3.2.2. This type of approach allows systems to better control the effects of a focus+context distortion on the drawing and mitigate the loss of aesthetic fulfillment.

5.3 Incremental Exploration

In *incremental exploration* visualization systems, large graphs are displayed in small portions instead overwhelming users with a single view of the entire graph [34, 36, 38, 73]. These system create a "visible" window for a drawing that allows users to explore subsections of the graph by moving the focus of this window. Since the entire graph no longer needs to be known or considered all at once, incremental exploration systems are often more responsive and computationally efficient.

There are many incremental exploration visualization systems [11, 23, 37, 38, 53]. Eklund, Sawers, and Zeiliger's NESTOR application creates subgraph views of the World Wide Web; users are shown subsections of the graph based on their web browsing histories [23]. Huang, Eades, and Wang's visualization system creates similar drawings but extend the visible subgraph to include neighboring web pages that the user might visit [38].

6 Interactive Spanning Tree Visualization

Our graph visualization system allows users to explore the structure and properties of a graph via multiple spanning-tree-based drawings. Given a graph, the system first displays a force-directed layout of the entire graph, using Eades' simulation model of charged particles for vertices and metal springs for edges [19]. A user then click on any vertex of the drawn graph. A spanning tree rooted at the selected vertex is extracted from the graph using breadth-first search. The system computes a drawing for the graph based on this spanning tree, and then uses animation to transition from the full graph drawing to this new drawing. Once this transition is complete, users can select a new root vertex for a different layout or return to the full graph view.

The screen captures in Figure 10 demonstrate how our visualization system generates a force-directed layout drawing for the full graph, and then transitions to a spanning tree drawing. The screen captures in Figure 11 demonstrate how our system generates a different spanning-tree-based drawing rooted at a different vertex for the same graph in Figure 10, and then transitions from the original drawing to the new drawing.

We now present our work on creating our graph visualization system in two parts. First, in Section 6.1 we present our graph drawing algorithm, which generates spanning-tree-based drawings for a graph. In Section 6.2, we present our animation algorithm, which generates smooth, continuous transitions between two graph drawings.

6.1 Context-Free Radial Layout

Given a graph G, a user-selected vertex $r \in V(G)$, and an initial graph drawing Γ for G, our graph drawing algorithm generates a new drawing of G based on a spanning tree rooted at r extracted from G. We call our algorithm "context free" because the placement of children, relative to the frame of reference of the parent, only depends on the parent's position. We use this term loosely; drawing schemes that also have this property, called shape grammars, have been studied more rigorously by other authors [55, 57].

Our visualization system creates a new drawing for a graph G with an initial drawing Γ in three stages. First it extracts a spanning tree T rooted at r from G using breadth-first search. Using this tree, the system calculates the graph's vertices' relative polar coordinates from their positions in Γ. Based on these initial positions, the system then calculates the vertices' relative polar coordinates in the new drawing. Instead of using concentric circles where one circle is used for positioning all the vertices for a given depth in the tree (Section 3.2.3), our algorithm creates drawings using a series of overlapping circles that we call *containment circles.* Each non-leaf vertex $v \in V(T)$ is given its own containment circle centered at v and only v's children are positioned on this circle. This approach enables the drawing algorithm to position sibling vertices close together and emphasize the parent-child relationships in the tree.

Before discussing the specifics of our new drawing method in Section 6.1.2, we first outline the parameters governing the construction of our graph

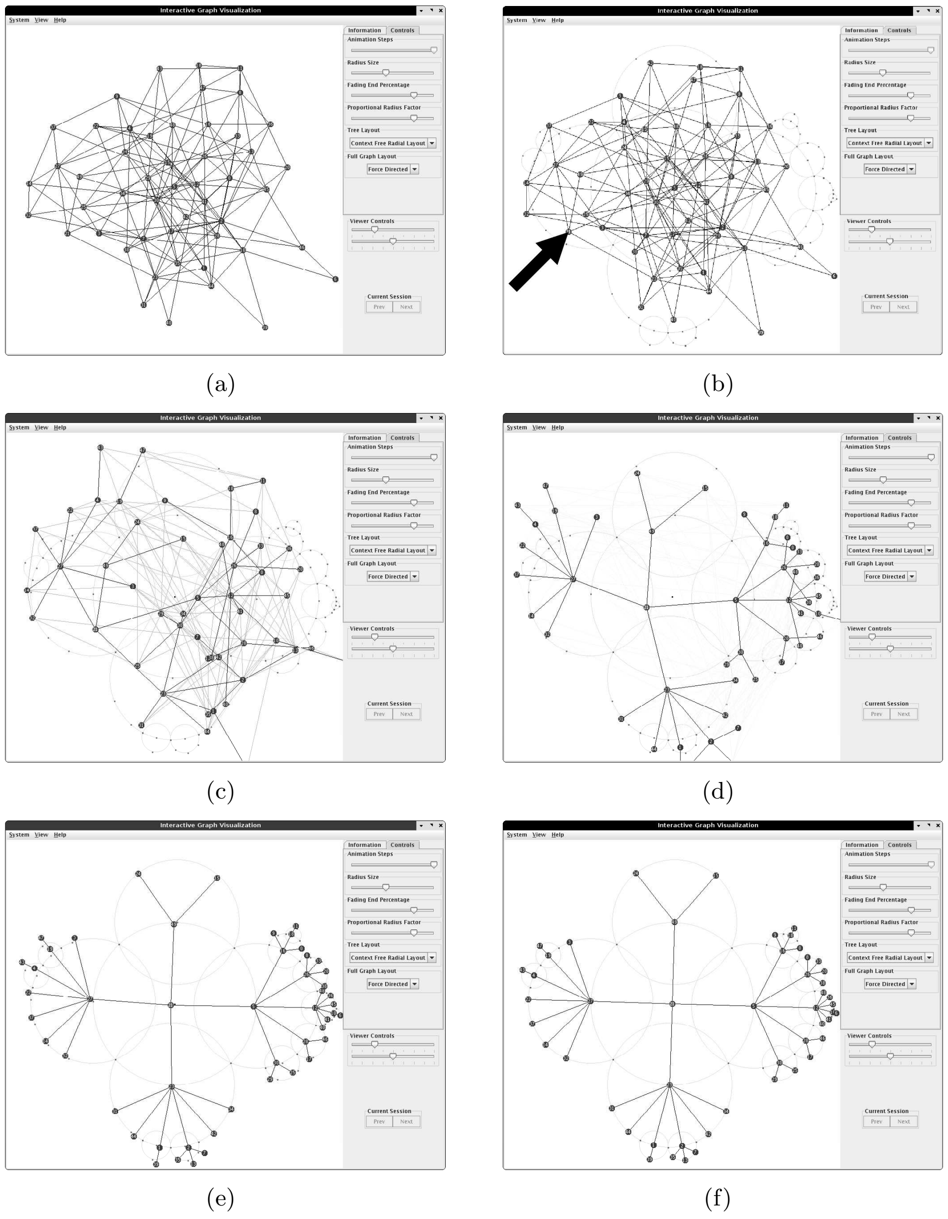
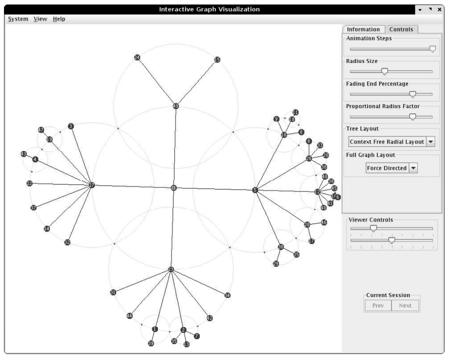
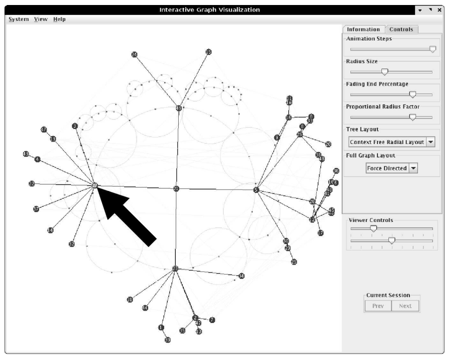


Figure 10: Our graph visualization system first generates a force-directed layout drawing of a graph with 50 vertices (Figure 10(a)). A user then selects a vertex (indicated by the arrow) to become the root of a new spanning-tree-based drawing for the graph (Figure 10(b)). The movement of the graph's vertices and edges is animated as the visualization system transitions from the original drawing to the new drawing (Figure 10(c) to Figure 10(e)). The animation sequence is complete when the vertices reach their final positions in the new layout (Figure 10(f)).



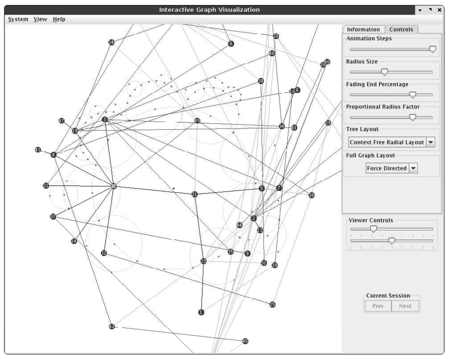
Animation Steps
Radius Size
Fading End Percentage
Proportional Radius Factor
Tree Layout: Context Free Radial Layout
Full Graph Layout: Force Directed

(a)



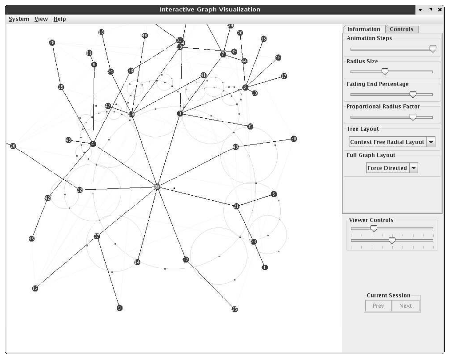
Animation Steps
Radius Size
Fading End Percentage
Proportional Radius Factor
Tree Layout: Context Free Radial Layout
Full Graph Layout: Force Directed

(b)



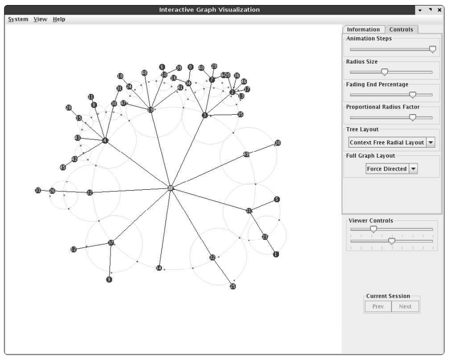
Animation Steps
Radius Size
Fading End Percentage
Proportional Radius Factor
Tree Layout: Context Free Radial Layout
Full Graph Layout: Force Directed

(c)



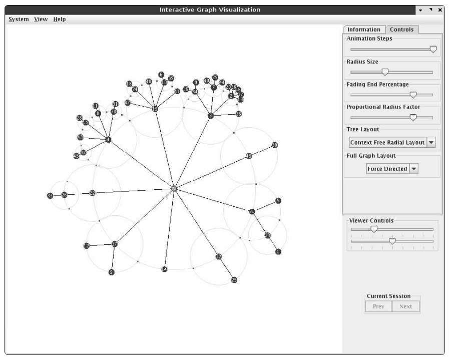
Animation Steps
Radius Size
Fading End Percentage
Proportional Radius Factor
Tree Layout: Context Free Radial Layout
Full Graph Layout: Force Directed

(d)



Animation Steps
Radius Size
Fading End Percentage
Proportional Radius Factor
Tree Layout: Context Free Radial Layout
Full Graph Layout: Force Directed

(e)



Animation Steps
Radius Size
Fading End Percentage
Proportional Radius Factor
Tree Layout: Context Free Radial Layout
Full Graph Layout: Force Directed

(f)

Figure 11: Using the same spanning tree drawing from Figure 10(f), a user selects a different vertex in the graph indicated by the arrow to become the root of a new spanning-tree-based drawing (Figure 11(b)). The movement of the graph's vertices and edges is animated as the visualization system transitions from the original drawing to the new drawing (Figure 11(c) to Figure 11(e)). The animation sequence is complete when the vertices reach their final positions in the new layout (Figure 11(f)).

Algorithm 3 *initialLayoutAngleAndDelta* $(T, v, \Upsilon_1, \Omega)$

```
$\kappa \Leftarrow$ The number of children of $v$
if ($v$ is the root of the tree $T$) then
  //The root is allocated $360°$ for its annulus wedge
  $\Psi \Leftarrow \frac{360}{\kappa}$
  $\vartheta \Leftarrow$ the sum of all of the root's children's angles in $\Upsilon_1$
  //$\Theta$ is the position of the root's first child
  $\Theta \Leftarrow \frac{1}{\kappa} \cdot \left(\vartheta - \Psi \cdot \frac{\kappa \cdot (\kappa+1)}{2}\right)$
else
  //Calculate the bounding angles $(\alpha, \beta)$ for $v$'s annulus wedge of size $\Omega$
  $\alpha \Leftarrow \theta + \left(360 - \frac{1}{2}\Omega\right)$
  $\beta \Leftarrow \alpha + \Omega$
  //$v$'s annulus wedge is divided evenly between its children
  $\Psi \Leftarrow \frac{(\beta - \alpha)}{\kappa}$
  //The initial angle $\Theta$ is the angle in the center of the first division
  $\Theta \Leftarrow \alpha + \frac{1}{2}\Psi$
end if
output $(\Theta, \Psi)$
```

Figure 12: Given a rooted tree T, a vertex $v \in V(T)$, and the initial drawing polar coordinate mapping Υ_1, the algorithm outputs the angle position Θ for v's first child and the delta angle Ψ that separates v's children on its containment circle. The angle Ω is the user-defined size of all non-root vertices' annulus wedges in the graph drawing.

drawings.

6.1.1 Graph Drawing Parameters

Graph drawing parameters describe the visual characteristics of a graph drawing (Section 3.1). The only convention our graph drawings follow is that a graph's edges are drawn as straight-line segments. The drawings adhere to three aesthetic goals: (1) minimize the number of edge crossings, (2) minimize the total angular difference between the root's children's positions from the initial drawing to the new drawing, and (3) maximize the angular resolution of parent-child edges. Our drawings conform to two constraints: (1) the root vertex is placed at the center of the drawing, and (2) vertices are equidistant to their parent vertex in the tree.

6.1.2 Layout Algorithm

Our context-free radial layout graph drawing algorithm computes a new drawing for a spanning tree extracted from a graph. Instead of using absolute Cartesian coordinates to position vertices, our algorithm computes a mapping $\Upsilon : V(T) \to (\theta, r)$, where (θ, r) are polar coordinates for a vertex $v \in V(T)$ and T is a spanning tree extracted from a graph G. Our drawing algorithm computes a mapping Υ_1 for the vertices' coordinates in the initial drawing Γ of a graph and a mapping Υ_2 for the vertices' coordinates in the new drawing being generated. We use a polar coordinate system because our animation algorithm computes animation sequences where vertices move on radial paths by interpolating coordinates from Υ_1 to Υ_2 (Section 6.2.2). One only needs Υ_2 to convert the vertices' polar coordinates to Cartesian coordinates to generate the new static drawing of the graph.

For a vertex $v \in V(T)$, v's parent vertex $p \in V(T)$, and the initial graph

Algorithm 4 $calculatePolarCoordsRelativeToParent\,(T, v, p, R, \Omega, \Gamma, \Upsilon_1, \Upsilon_2)$

```
if ($v$ is the root of the tree $T$) then
  $\Upsilon_1(v) \Leftarrow ((\Gamma(v))_\theta, (\Gamma(v))_r)$
  $\Upsilon_2(v) \Leftarrow ((\Gamma(v))_\theta, 0)$
end if
if ((the number of children of $v$ in $T$) $> 0$) then
  //Calculate $v$'s childrens' polar coordinates relative to $v$'s position in $\Gamma$
  for all (children $c$ of $v$ in $T$) do
    $\Upsilon_1(c) \Leftarrow ((\Gamma(c) - \Gamma(v))_\theta - (\Upsilon_1(v))_\theta, (\Gamma(c) - \Gamma(v))_r)$
  end for
  //Calculate the initial angle $\Theta$ and the delta angle $\Psi$ for $v$'s children
  $(\Theta, \Psi) \Leftarrow initialLayoutAngleAndDelta\,(T, v, \Upsilon_1, \Omega)$
  if ((the number of children of $v$ in $T$) $= 1$) then
    $\lambda \Leftarrow \frac{R}{2}$
  else
    $\lambda = 2R \cdot \sin\left(\frac{\Psi}{2}\right)$
  end if
  //Calculate $v$'s childrens' positions in the new drawing $\Upsilon_2$
  for all (children $c$ of $v$ in $T$, chosen according to the counterclockwise rotational
  ordering in $\Upsilon_1$ of the children of $v$) do
    $\Upsilon_2(c) \Leftarrow (\Theta, R)$
    $(\Upsilon_1, \Upsilon_2) \Leftarrow calculatePolarCoordsRelativeToParent\,(T, c, v, \lambda, \Omega, \Gamma, \Upsilon_1, \Upsilon_2)$
    $\Theta \Leftarrow \Theta + \Psi$
  end for
end if
output $(\Upsilon_1, \Upsilon_2)$
```

Figure 13: Given a rooted tree T, a vertex $v \in V(T)$, v's parent vertex $p \in V(T)$, the radius R of v's containment circle, and the initial graph drawing Γ, the algorithm calculates the relative radial coordinates of every child vertex c of v in the initial drawing mapping Υ_1 and the new drawing mapping Υ_2. For any point $\vec{p}$ in a Cartesian coordinate system, let $(\vec{p}_\theta, \vec{p}_r)$ denote the polar coordinates of $\vec{p}$. The angle Ω is the user-defined size of all non-root vertices' annulus wedges in the graph drawing.

Algorithm 5 *initialLayoutAngleAndDelta* $(T, v, \Upsilon_1, \Omega)$

```
$\kappa \Leftarrow$ The number of children of $v$
if ($v$ is the root of the tree $T$) then
    //The root is allocated 360° for its annulus wedge
    $\Psi \Leftarrow \frac{360}{\kappa}$
    $\vartheta \Leftarrow$ the sum of all of the root's children's angles in $\Upsilon_1$
    //$\Theta$ is the position of the root's first child
    $\Theta \Leftarrow \frac{1}{\kappa} \cdot \left(\vartheta - \Psi \cdot \frac{\kappa \cdot (\kappa+1)}{2}\right)$
else
    //Calculate the bounding angles $(\alpha, \beta)$ for $v$'s annulus wedge of size $\Omega$
    $\alpha \Leftarrow \theta + \left(360 - \frac{1}{2}\Omega\right)$
    $\beta \Leftarrow \alpha + \Omega$
    //$v$'s annulus wedge is divided evenly between its children
    $\Psi \Leftarrow \frac{(\beta - \alpha)}{\kappa}$
    //The initial angle $\Theta$ is the angle in the center of the first division
    $\Theta \Leftarrow \alpha + \frac{1}{2}\Psi$
end if
output $(\Theta, \Psi)$
```

Figure 14: Given a rooted tree T, a vertex $v \in V(T)$, and the initial drawing polar coordinate mapping Υ_1, the algorithm outputs the angle position Θ for v's first child and the delta angle Ψ that separates v's children on its containment circle. The angle Ω is the user-defined size of all non-root vertices' annulus wedges in the graph drawing.

drawing Γ for T, the *calculatePolarCoordsRelativeToParent* algorithm (page 34) computes the positions of v's children in the new drawing in four parts: (1) if v is the root of T, position v at the center of the drawing, (2) calculate v's children's relative polar coordinates in Υ_1 based on the children's positions in Γ, (3) allocate a containment circle and an annulus wedge to position v's children in the new drawing, and (4) calculate v's children's relative polar coordinates in Υ_2 such that they are positioned evenly on v's containment circle within the bounds of v's annulus wedge.

First, the algorithm positions the root at the center of the new drawing at $(0, 0)$. The root's coordinates in Υ_1 are derived by converting $\Gamma(v)$ to absolute polar coordinates. The root's polar coordinate radius in Υ_2 is set at zero, but its polar coordinate angle is the same as in Υ_1. This ensures that the root moves on a straight-line path towards the origin of the drawing during the animated transitions in our system (Section 6.2).

After the root's coordinates are calculated in both Υ_1 and Υ_2, the algorithm calculates v's children vertices' polar coordinates in Υ_1 based on their positions in Γ. For a child vertex c of v, c's polar coordinate radius is the Euclidean distance from $\Gamma(c)$ to $\Gamma(v)$ and its angle is relative to v's position in Γ.

Next, the system calculates the initial angle Θ for the first child of v, and the delta angle Ψ separating v's children on v's containment circle in the new drawing. For a tree T, a vertex $v \in V(T)$, the initial drawing mapping Υ_1 for T, and the user-defined size of all non-root annulus wedges Ω, the *initialLayoutAngleAndDelta* algorithm (page 35) calculates relative position of v's annulus wedge and outputs the angle set (Θ, Ψ).

If v is the root of the tree, v's annulus wedge is the entire angle space of the drawing (360°). The root's initial angle Θ for its children is calculated such that the angular difference from the root's children's positions in Υ_1 to their positions in the new drawing is minimized. Although other layout configurations may result in lower overall movement for the entire graph, our algorithm only considers minimizing the rotational movement of the root's children.

If v is not the root of the tree, v's annulus wedge is a portion of v's containment circle. The user-defined angle Ω specifies the size of v's annulus wedge, and the angles α and β denote where on v's containment circle the wedge begins and ends. In order to adhere to our aesthetic goal of maximizing the angular resolution of parent-child edges, Ω is always less than or equal to $180°$ (Ω is fixed at $180°$ in the examples in Figure 15 and Figure 16). The size of Ω influences the visual characteristics of the graph drawing: smaller annulus wedges produce tighter and more narrow subtree layouts, while larger wedges causes trees to fan out and use more space. The center point of v's annulus wedge is the center of the arc on v's containment circle that is outside of v's parent's circle. The size of v's annulus wedge is always less than the size of this outer arc, and thus v's children are not positioned at overlapping circles' intersection points (otherwise vertices would occlude other vertices in neighboring circles). The *initialLayoutAngleAndDelta* algorithm calculates the angles α and β using the angle from v to its parent as the relative $0°$ in space on v's containment circle.

The vertex v's annulus wedge is now divided into equal-sized portions for each of v's children. Each child vertex is allocated the same amount of space on v's containment circle regardless of the size of its subtree. The initial angle Θ is the center angle for the first subdivision of v's annulus wedge. The delta angle Ψ is the angular size of these annulus wedge subdivisions. In the case where v has only one child, that child is positioned at center of

v's entire annulus wedge.

Next, the *calculatePolarCoordsRelativeToParent* algorithm computes the radius λ of the containment circles for the next level of children in the subtree. This radius λ is passed as the input for R in the next recursive iteration of the drawing algorithm. If the number of children of $v > 1$, then λ is the length of the chord from Θ to $\left(\Theta + \frac{\Psi}{2}\right)$ on v's containment circle. If the number of children of $v = 1$, then λ is $\frac{R}{2}$; this ensures that the containment circles in the drawing get progressively smaller as vertices are positioned further away from the root.

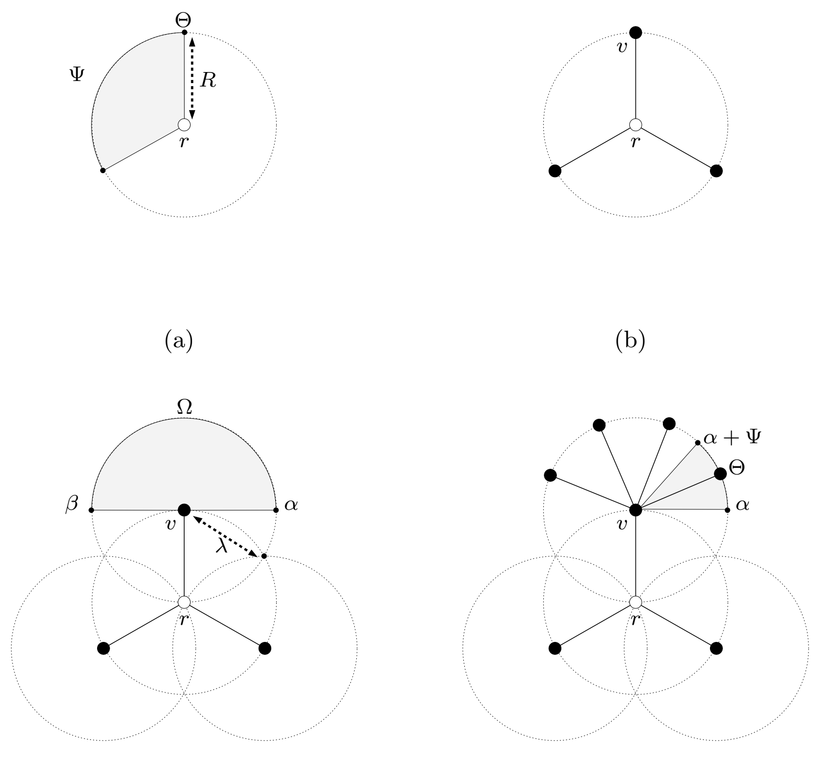
The algorithm now iterates through v's children based on their counter-clockwise rotational ordering in Υ_1, and assigns each child a position in Υ_2. Like their coordinates in Υ_1, v's children's positions in Υ_2 are relative to v's position, but are now based on v's new position in Υ_2. For each child vertex c of v, c's polar coordinate radius in Υ_2 is R, which is passed as input to the algorithm, and c's angle in Υ_2 is Θ, which is incremented by Ψ for each child. If v is the root of the tree, then the initial value for R is defined by the user.

The drawing algorithm continues recursively down each subtree in a depth-first fashion until all the vertices' polar coordinates in Υ_1 and Υ_2 are calculated. The *calculatePolarCoordsRelativeToParent* algorithm runs in $O(n)$ time.

Figure 15 and Figure 16 provide a visual example of how our drawing algorithm constructs a new spanning-tree-based drawing for a graph.

6.2 Animated Tree Transition

In our interactive graph visualization system, we provide animated transitions to aid users' exploration of multiple drawings for a graph. We now



(a) (b)

(c) (d)

Figure 15: The above diagram illustrates how our graph drawing algorithm constructs a new drawing for a tree T rooted at r. In Figure 15(a), the root is first placed at the center of the drawing along with its containment circle with a radius of R. The root's annulus wedge is divided into three equal portions of size Ψ and its first child is positioned at Θ. In Figure 15(b), the root's children are positioned on its containment circle. Next, in Figure 15(c) each of the root's children is allocated a separate containment circle with a radius of λ. The algorithm then allocates space in the drawing to position v's children. v's annulus wedge of size Ω is centered on the arc of v's circle that is outside of the root's circle. The angles α and β are relative to v's position. In Figure 15(d), v's annulus wedge is divided into four equal parts of size Ψ. Each child of v is positioned in the center of one v's annulus wedge subdivisions starting at Θ.

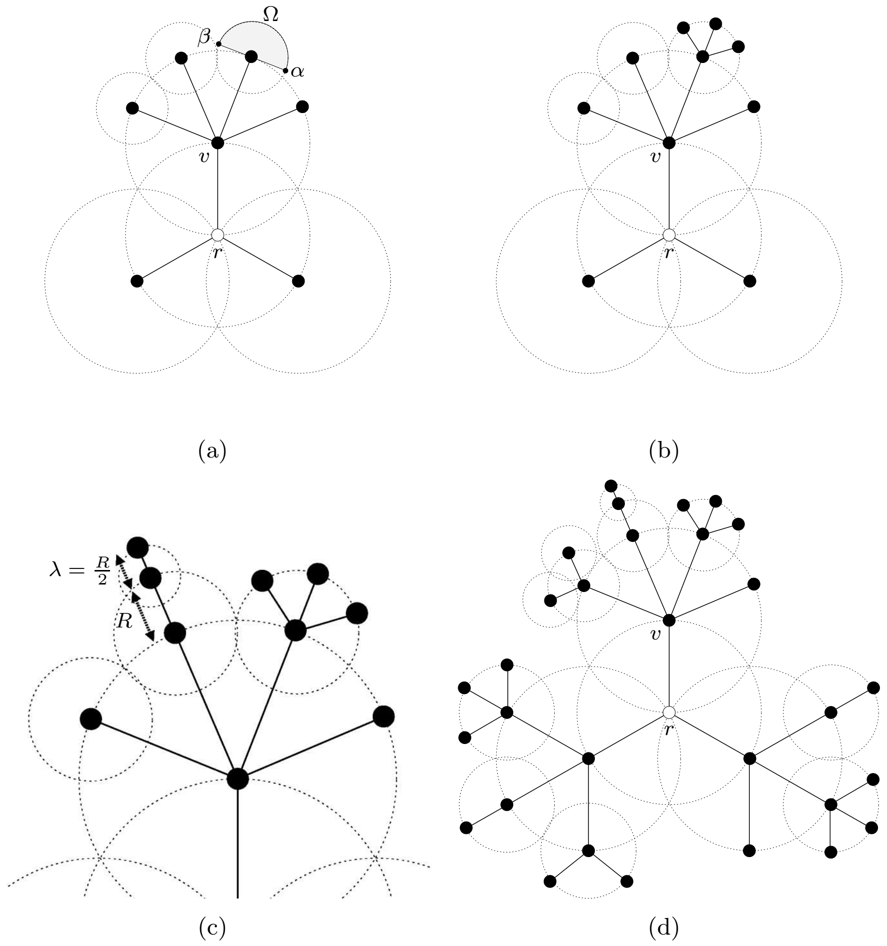


Figure 16: (Continued from Figure 15) In Figure 16(a) and Figure 16(b), the drawing algorithm continues down the subtree rooted at v allocating containment circles and annulus wedges for descendant vertices. In Figure 16(c), the radius λ is half the size of that vertex's parent's containment circle. The algorithm positions the rest of the graph's vertices, resulting in the final drawing shown in Figure 16(d).

present our graph animation algorithm that computes a sequence of frames to transition any graph drawing to one of our spanning-tree-based drawings.

In Section 6.2.1, we first outline the particular goal we seek to accomplish with our graph animation algorithm. In Section 6.2.2, we discuss the implementation specifics of our algorithm. In Section 6.2.3, we discuss two auxiliary visual cues we incorporate into our system to further help users maintain continuity during transitions.

6.2.1 Animation Goal

The principles of our animation algorithm are guided by previous graph drawing aesthetic studies that suggest users are better able to comprehend drawings that minimize the number of edge crossings [39, 58, 70]. Although we are not aware of research that measures the effectiveness of drawing aesthetics as applied to graph animation, we believe that the results of these studies are certainly applicable to our work. Thus, the main goal in our transitions from one drawing to another is to minimize the number of crossings.

This goal is difficult to achieve because it is not a trivial task to generate an animation sequence for tree drawings with no edge crossings. For example, the radial layout animation algorithm in Section 4.2.3 produces crossings even when transitioning between two drawings of the same tree. Rather than try to prevent all crossings, our animation algorithm is designed to prevent two specific types: (1) crossings between sibling vertices, and (2) crossings between the edge of a vertex to its parent with one of its edges to its children. Our visualization system eliminates these crossings because vertices' positions are calculated relative to their parent and the rotational ordering of sibling vertices is preserved from their positions in initial drawing.

In preventing these crossings, our algorithm does not calculate movements simply by choosing the shortest path to move a vertex from one point to another. We believe that the benefits of avoiding an edge crossing outweigh any increased movement in the animation.

6.2.2 Animation Algorithm

Our graph animation algorithm computes a series of frames that transition a graph from an initial layout to a new drawing generated by our drawing algorithm in Section 6.1.2. Given rooted tree T, a vertex $v \in V(T)$, v's parent vertex $p \in V(T)$, a time step s in the animation sequence, and an initial drawing mapping Υ_1 and a new drawing mapping Υ_2 for T, the *transition* algorithm (page 43) computes the positions of v's children in the animation sequence at time step s, and outputs a graph drawing frame Γ_s. This frame is calculated by interpolating each vertex's relative polar coordinates from Υ_1 to Υ_2 using the parent's position at the current time step as a frame of reference. To generate all the intermediate frames for the animation, one would invoke our algorithm from time steps 1 to S, where S is the last frame in the animation sequence.

The algorithm first calculates the straight-line path movement of the root vertex from its position in Υ_1 to the center of the drawing at $(0, 0)$. The difference between this straight-line movement and the radial movement of all other vertices creates a visual contrast that allows users to easily identify the new root of the tree.

Now the animation algorithm computes v's children's positions in Γ_s. For a child vertex c of v, the algorithm first calculates c's relative polar coordinates (θ, r) by interpolating from $\Upsilon_1(c)$ to $\Upsilon_2(c)$ using Δ as the in-

Algorithm 6 *transition* $(T, v, p, s, \Upsilon_1, \Upsilon_2, \Gamma_s)$

```
$\Delta \Leftarrow \frac{s}{S}$
**if** ($v$ is the root of the tree $T$) **then**
    //The root moves on a straight-line path to $(0, 0)$
    $(\theta, r) \Leftarrow \Upsilon_1(c)$
    $(x, y) \Leftarrow ((r \cdot (1 - \Delta)) \cdot \cos(\theta), (r \cdot (1 - \Delta)) \cdot \sin(\theta))$
    $\Gamma_s(v) \Leftarrow (x, y)$
    $\varphi \Leftarrow ((0, 0) - \Gamma_s(v))_\theta$
**else**
    $(x, y) \Leftarrow \Gamma_s(v)$
    $\varphi \Leftarrow (\Gamma_s(p) - \Gamma_s(v))_\theta$
**end if**
**for all** (children $c$ of $v$ in $T$) **do**
    $(\Theta_1, R_1) \Leftarrow \Upsilon_1(c)$
    $(\Theta_2, R_2) \Leftarrow \Upsilon_2(c)$
    //Calculate $c$'s relative polar coordinates for this time-step
    $\theta \Leftarrow (\Theta_1 \cdot (1 - \Delta)) + (\Theta_2 \cdot \Delta)$
    $r \Leftarrow (R_1 \cdot (1 - \Delta)) + (R_2 \cdot \Delta)$
    //Convert $(\theta, r)$ to absolute Cartesian coordinates for frame $\Gamma_s$
    $\Gamma_s(c) \Leftarrow (x + (r \cdot \cos(\theta + \varphi)), y + (r \cdot \sin(\theta + \varphi)))$
    //Calculate animation frames for $v$'s children
    **if** ((the number of children of $c$ in $T$) $> 0$) **then**
        $\Gamma_s = transition(T, c, v, s, \Upsilon_1, \Upsilon_2, \Gamma_s)$
    **end if**
**end for**
**output** $\Gamma_s$
```

Figure 17: Given a rooted tree T, a vertex $v \in V(T)$, v's parent vertex $p \in V(T)$, the current time step s in the animation, and an initial drawing mapping Υ_1 and a new drawing mapping Υ_2, the algorithm calculates the positions of v's children at time step s and outputs an intermediate frame Γ_s. For any point $\vec{p}$ in a Cartesian coordinate system, let $(\vec{p}_\theta, \vec{p}_r)$ denote the polar coordinates of $\vec{p}$. Let Δ be the interpolation factor at time step s. Let S be the total number of frames in the animation sequence.

terpolation factor at time step s. These polar coordinates are then offset by the reference angle φ and v's position in Γ_s. If v is the root of the tree, then φ is the angle from $\Gamma_s(v)$ to the center of the drawing at $(0, 0)$. If v is not the root of the tree, then φ is the relative angle of v's parent in Γ_s using v as a point of reference. If c has children of its own, the algorithm recursively invokes itself to calculate the positions for the vertices in the subtree rooted at c.

The algorithm finishes once positions for all the vertices in the graph are calculated for Γ_s. The *transition* algorithm runs in $O(n)$ time.

Figure 18 and Figure 19 provide a step-by-step example of the how animation sequences are calculated by our animation algorithm.

6.2.3 Auxiliary Visual Enhancements

A graph drawing transition may visually overwhelm users with information [29]. Although good graph animation alleviates many problems, additional visualization techniques can further help users maintain their mental continuity between drawings. We now summarize the auxiliary visual enhancements we use to improve the usefulness of the transitions created by our animation algorithm.

First, we adopt the slow-in, slow-out timing used by Yee et al.'s Gnutellavision system for the movements of vertices [73]. Variable speed approaches such as this provide adequate visual constancy for users. This type of movement mimics the acceleration and deceleration of massive objects in the real physical world, which research suggests that the human brain is pre-disposed to understand more easily [51]. To achieve this effect, our animation algorithm calculates the interpolation factor Δ from the curve of the arctangent function, as shown in the equation below. Let s be the current time step

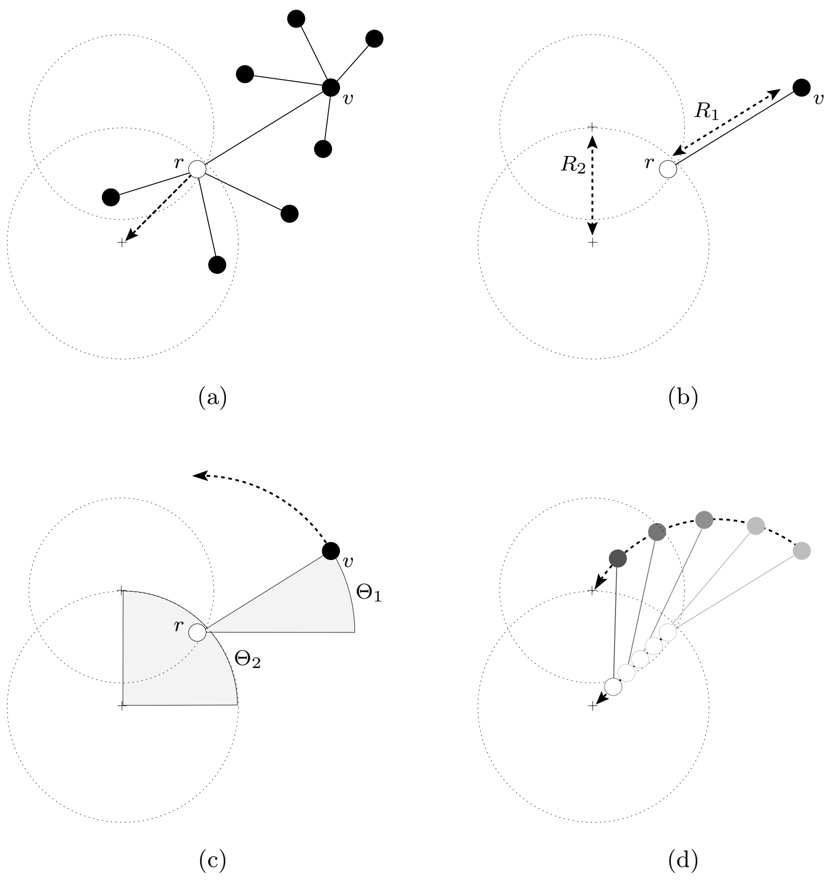


Figure 18: The above diagram is an example of how our graph animation algorithm generates an animated transition for a tree T rooted at r to a new graph drawing. The algorithm first calculates the straight-line movement of the root, shown in Figure 18(a), from its initial position to the center of the drawing (denoted by the cross). The movement of vertex v is derived by interpolating from the radii R_1 to R_2, shown in Figure 18(b), and from the angles Θ_1 to Θ_2, shown in Figure 18(c). v's movement is also based on its parent's position at each time step in the animation; as v's polar coordinates are interpolated it moves relative to the root, as shown Figure 18(d).

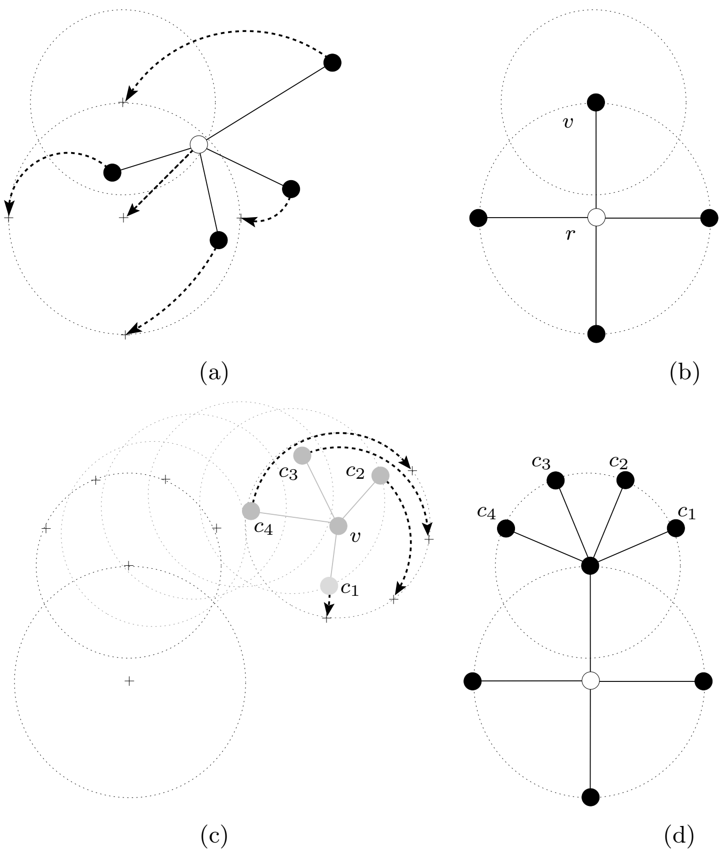


Figure 19: (Continued from Figure 18) The root's children vertices move to their positions in the new drawing, shown in Figure 19(a) and Figure 19(b). The movements of v's children vertices c_1, c_2, c_3, and c_4's are derived through the interpolation of their polar coordinates relative to v, shown in Figure 19(c) and Figure 19(d).

in the animation and let S be the total number of steps in the animation sequence:

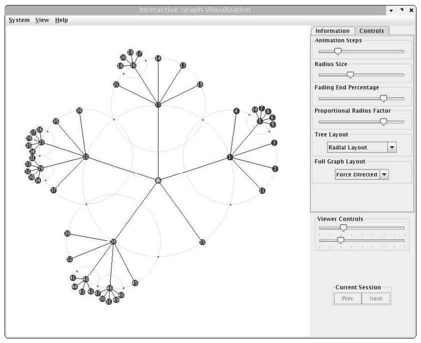
$$\Delta \Leftarrow \frac{1}{2}\left(\frac{\tan^{-1}\left(\frac{s\cdot 10}{S}-5\right)}{\tan^{-1}(5)}\right)+\frac{1}{2} \tag{1}$$

Using this method of interpolation provides a "slow-start" movement in the animation, allowing users to anticipate the general paths of vertices in the ensuing transition. The movement of the graph accelerates to the midpoint of the transition, and then decelerates as vertices reach their final positions in the new drawing. With this timing, users are presented with a transition that seems neither too fast nor too slow.

The second visual enhancement is the fading in and out of graph elements used by Friedrich and Eades [29]. Our implementation differs from Friedrich and Eades in that we fade elements during the transition, rather than before and after. We believe that including transient edges during the animation may allow users to study a graph more carefully; by having graph elements gradually materialize and disappear as the graph moves, users also may be able see the structural differences between two graph drawings with greater ease.

The calculation of the fading factor during an animation sequence is different for the two types of transition scenarios in our visualization system. When transitioning from a spanning tree drawing back to the full graph, the fading factor is based on a fixed delta for a finite time period. Because our force-directed algorithm implementation is non-deterministic, we are unable to fade relative to when the simulation will reach equilibrium.

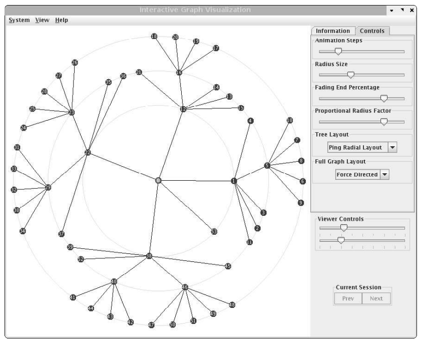
The fading factor for spanning-tree-to-spanning-tree transitions in our animation algorithm is relative to the current time step in the animation sequence. However, this fading factor is not always equivalent to the algo-



Tree Layout: Radial Layout

Full Graph Layout: Force Directed

(a)



Tree Layout: Ping Radial Layout

Full Graph Layout: Force Directed

(b)

Figure 20: Two screen captures of graph drawings for the same tree: the system in Figure 20(a) uses our visualization scheme, and the system in Figure 20(b) uses Yee et al.'s visualization scheme from their Gnutellavision application [73].

rithm's interpolation factor Δ; the fading process of edges can be finished before the graph stops moving. Edges fading out during a transition create superfluous crossings because they are not taken into consideration by our animation algorithm. Removing fading edges at an animation sequence time step before the transition is completed reduces the number of crossings caused by these edges.

7 Experimental Analysis

We now describe a series of experiments that test whether our context-free radial layout graph drawing and animation algorithms visualize graphs better than Yee et al.'s drawing and animation algorithms from their Gnutellavision application. We test both systems on randomly generated graphs and spanning trees extracted from these graphs using randomly selected root vertices. Figure 20 shows screen captures of graph drawings generated by our and Yee et al.'s algorithms.

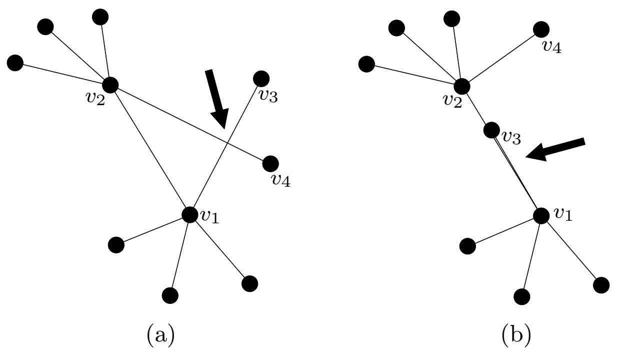


Figure 21: An edge crossing occurs when the two nonadjacent edges $\{v_1, v_3\}$ and $\{v_2, v_4\}$ as in Figure 21(a) intersect, or when the two edges $\{v_1, v_3\}$ and $\{v_1, v_2\}$ overlap as in Figure 21(b).

7.1 Measurements

Our experiments measure two aspects of interactive graph visualization: edge crossings and sibling edge lengths.

7.1.1 Edge Crossings

An edge crossing occurs when either two nonadjacent edges intersect at a single point, or when two edges overlap (see Figure 21). Our experiments measure the number of crossings that occur during a transition between two graph drawings. Since the graph layout is changing during these transitions, if two edges remain crossed over multiple animation frames we count only one crossing for that unique pair of edges.

We divide edge crossings into two categories: transient crossings and final layout crossings. A crossing is *transient* if at least one of the edges is fading out from the drawing during the animation. A *final layout* crossing occurs when both edges are part of the final drawing that the system is transitioning to.

7.1.2 Sibling Edge Lengths

Our second measurement is the length of edges for sets of sibling vertices in a graph. For a given graph, we calculate the mean and standard deviation edge length for all sets of sibling vertices and then calculate the mean of those standard deviations. Given a tree T and a graph drawing Γ of T, for all vertices $v \in V(T)$ where the number of children of $v > 0$, the equation below formulates the mean standard deviation $\overline{\sigma}$ of sibling edge lengths in Γ (over all non-empty groups of siblings). For a given vertex v, let κ_v be the number of children of v and let μ_v be the mean Euclidean distance between v and each of its children in Γ. For two vertices $u, v \in V(T)$, let $d_\Gamma(u, v)$ be the Euclidean distance from $\Gamma(u)$ to $\Gamma(v)$. Let N be the number of non-leaf children in T.

$$\overline{\sigma} = \frac{1}{N} \cdot \sum_{\substack{v \in V(T) \\ \text{number of} \\ \text{children of } v > 0}} \left(\sqrt{\frac{1}{\kappa_v} \cdot \sum_{\substack{\text{children} \\ c \text{ of } v}} \left(d_\Gamma(v, c) - \mu_v\right)^2} \right) \tag{2}$$

In the drawings produced by our algorithm, the lengths of edges for each set of sibling vertices are always equal (sibling vertices are equidistant to their parent). Thus, the standard deviation for sibling edge lengths in our drawings is always zero. This is an important feature of our visualization scheme as it allows users to perceive the depth of vertices from the root. This is not the case in the drawings created by Yee et al.'s algorithm.

7.2 Experiments

Table 1 provides a summary our experiments. In experiments 1–3, we measure the number of edge crossings that occur in a graph during an animation sequence. Experiment 1 measures transitions between drawings of two trees

Experiment 1 – Isomorphic Tree Transitions	
Question:	How well can the system transition a tree to different vertex-centric drawings?
Starting Condition:	Tree drawing
Ending Condition:	Isomorphic drawing of the same tree
Measurements:	Number of unique edge crossings during transition

Experiment 2 – Spanning-Tree-to-Spanning-Tree Transitions	
Question:	How well can the system transition from a spanning tree drawing to a different spanning tree drawing?
Starting Condition:	Spanning tree drawing
Ending Condition:	Spanning tree drawing
Measurements:	Number of unique edge crossings during transition

Experiment 3 – Full-Graph-to-Spanning-Tree Transitions	
Question:	How well can the system transition from the full graph drawing to a spanning tree drawing?
Starting Condition:	Force-directed drawing of the full graph
Ending Condition:	Spanning tree drawing
Measurements:	Number of unique edge crossings during transition

Experiment 4 – Spanning Tree Sibling Edge Lengths	
Question:	What is the edge-length standard deviation for sets of siblings in a graph drawing?
Starting Condition:	Spanning tree drawings from Experiment 3
Ending Condition:	n/a
Measurements:	Mean standard deviation of sibling edge lengths

Table 1: Summary of the experiments conducted with our graph drawing and animation algorithms, and with Yee et al.'s radial graph drawing and animation algorithms from their Gnutellavision visualization system.

Algorithm 7 *generateRandomGraph* (n, p)

```
for all $e \in E(K_n)$ do
    pick $x \in [0, 1]$
    if $x < p$ then
        add $e$ to $E(G)$
    end if
end for
ouput $G$
```

Figure 22: Given a natural number n, and a probability p, the algorithm outputs a graph G of order n using the Erdös-Rényi random graph generation model.

T_1 rooted at r_1 and T_2 rooted at r_2, where $r_1 \neq r_2$ and $E(T_1) = E(T_2)$. This is a special case of the spanning-tree-to-spanning-tree transition because crossings are avoidable; the animation should produce no crossings since the graph's edge set does not change between drawings and trees are always planar. Experiment 2 is similar to experiment 1 in that it measures transitions from a drawing of a spanning tree T_1 rooted at r_1 to a drawing of a spanning tree T_2 rooted at r_2, where $r_1 \neq r_2$, except now $E(T_1) \neq E(T_2)$. Experiment 3 measures transitions from a full graph drawing generated by a force-directed algorithm to a spanning-tree-based drawing root at a randomly selected vertex. Note that Yee et al.'s original visualization scheme does not use a force-directed layout to display the full graph and that this transition scenario is specific to our experimental system. Lastly, experiment 4 measures the edge lengths of sets of sibling vertices in the final spanning tree drawings produced in experiment 3.

7.3 Methodology

For each experiment, we test our and Yee et al.'s drawing and animation algorithms on a set of randomly generated graphs. For each natural number n ranging from 30 to 100 (inclusive), we conduct one trial of each experiment on 10 distinct graphs of order n. Each graph is generated according to the

Experiment Parameters	
Number of Vertices	30 - 100
Trials per Graph Order	10
Edge Connectivity Probability	10%
Number of Animation Steps	150
Animation Fading Step	150
Inner Circle Radius	250
Radius Increment	100
Annulus Wedge Size	180°

Table 2: The visualization system parameters used during the experiments.

Erdös-Rényi random graph generation model shown in Figure 22 [24]. We fix the probability p of an edge existing between any two vertices at 0.1 (10%).

Along with each graph, a series of root vertices are selected randomly that are used to extract spanning trees in each trial run. Experiments 1 and 2 require two distinct root vertices, while experiment 3 only needs a single root vertex.

After all the trials for an experiment are complete, we calculate the mean of the results for each set of 10 trials having graphs of the same order. We ensure that both visualization schemes conduct the same trial run using the same starting layout configuration for the given graph. Because our force-directed algorithm implementation does not guarantee that a graph is always drawn the same (there is small randomness factor included in the repulsion forces whenever one vertex occludes another), an initial full graph layout for each trial is computed and stored on disk prior to testing.

7.4 Testing Environment Constants

Table 2 lists the various parameters used during testing. These settings produce graph drawings that are readable and animation sequences that are smooth and useful.

7.5 Results

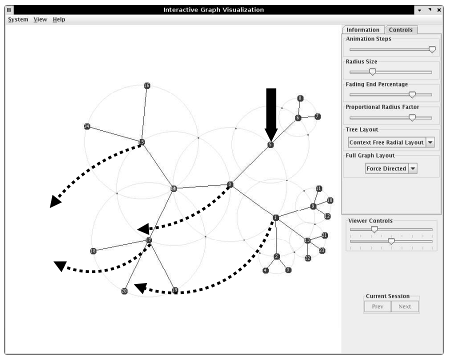
We now discuss the results from each of the four experiments.

7.5.1 Experiment 1 – Isomorphic Tree Transitions

We measure the number of edge crossings that occur during a transition between two drawings of the same tree. First, we select a random vertex as the root and have the system transition from a force-directed drawing of the tree to a drawing generated by one of the algorithms being tested. We then transition to another drawing with a different vertex randomly chosen as the new root.

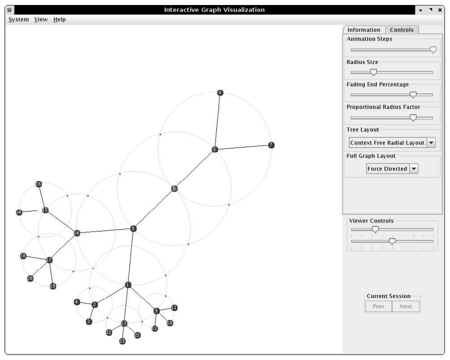
As shown clearly in Figure 25, our algorithms produce zero crossings while Yee et al.'s algorithms produce many. Our system achieves this crossing-free movement by scaling different parts of the graph; sibling vertices move together as rigid-objects as the parent's containment circle expands or contracts moving towards the new layout (see Figure 23 on page 55). This movement effect is remarkably similar to the clustered animation by Friedrich and Eades (Section 4.2.2).

The edge crossings produced by Yee et al.'s algorithms occur for two possible reasons (see Figure 24 on page 55). Yee et al.'s animation algorithm constrains each vertex to move along the shortest radial path to its destination, even if this results in additional crossings. Yee et al.'s drawing algorithm also mandates that the direction of the edge from the new root vertex to its parent in the previous drawing is preserved in the new drawing. Such a constraint can cause vertices to rotate around the origin of the drawing because the algorithm creates a dramatically different layout for the tree. Our general impression is that this occurs most often when a leaf vertex is chosen for the root of the new drawing and the previous root



Tree Layout: Context Free Radial Layout
Full Graph Layout: Force Directed

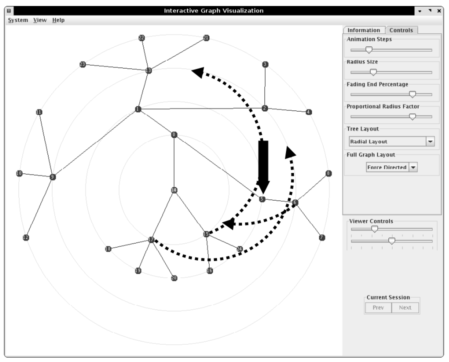
(a)



Tree Layout: Context Free Radial Layout
Full Graph Layout: Force Directed

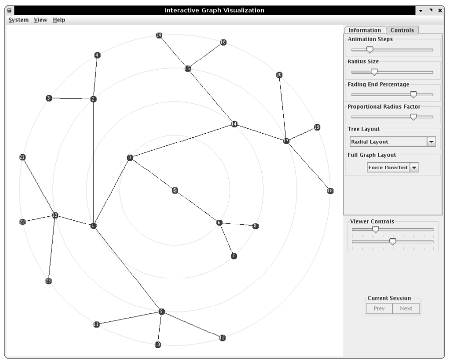
(b)

Figure 23: In this example, our visualization scheme transitions between two different drawings of the same tree. The root vertex for the drawing in Figure 23(b) is indicated by the solid black arrow in Figure 23(a). The system moves to Figure 23(b) without any edge crossings; the animation sequence scales and translates vertices' containment circles along radial paths (indicated by the dashed arrows) to the new drawing.



Tree Layout: Radial Layout
Full Graph Layout: Force Directed

(a)



Tree Layout: Radial Layout
Full Graph Layout: Force Directed

(b)

Figure 24: Using the same graph and root vertices as in Figure 23, Yee et al.'s visualization scheme transitions between two different drawings of the same tree. The root vertex for the drawing in Figure 24(b) is indicated by the solid black arrow in Figure 24(a). Yee et al.'s algorithms produce edge crossings even though the initial drawing and the new drawing are of the same tree. The dashed arrows indicate the general path of the vertices that cause the edge crossings during the transition.

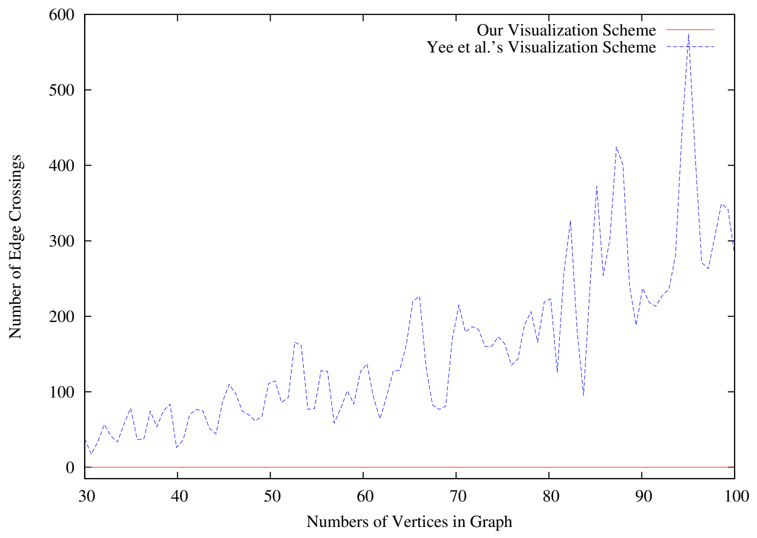


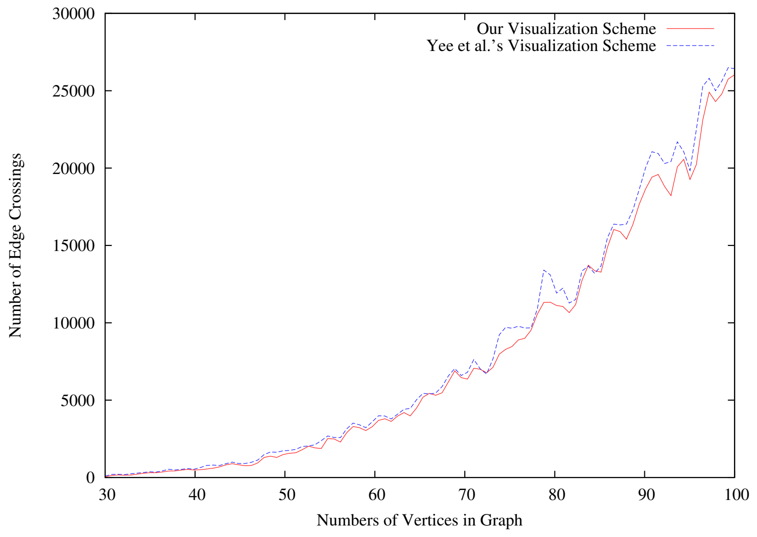
Figure 25: Experiment 1 – Isomorphic Tree Transitions
Our visualization scheme produces no edge crossings when transitioning between drawings of the same tree, while Yee et al.'s system produces many.

was more centrally located in the tree from the previous drawing. If the new root vertex and the previous root vertex are both non-leaf vertices, Yee et al.'s algorithms morph more simply, rather like ours.

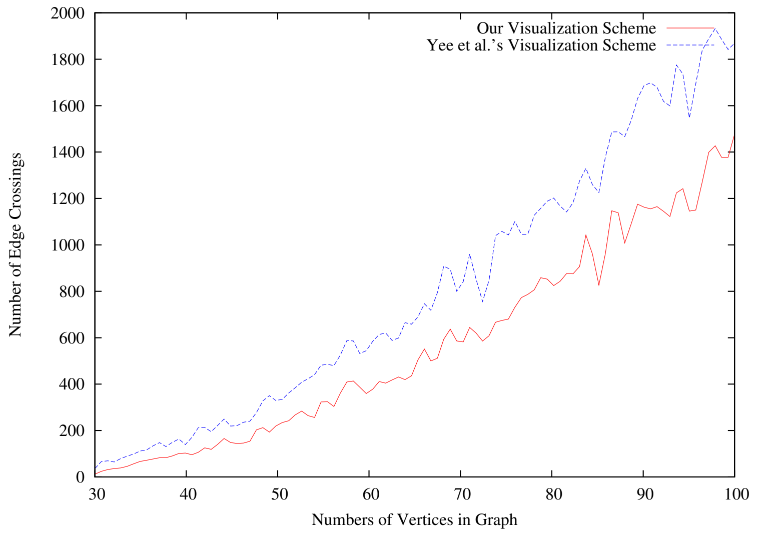
7.5.2 Experiment 2 – Spanning-Tree-to-Spanning-Tree Transitions

In experiment 2, we first transition from a full graph drawing to a spanning-tree-based drawing rooted a randomly selected vertex. We then transition from this spanning tree drawing to another spanning-tree-based drawing rooted at different vertex. We count the number of edge crossings that occur only in the second transition.

Figure 26 (page 57) shows that our algorithms produce 30% fewer final layout crossings and 6% fewer total crossings than Yee et al.'s algorithms over all experiment trials. With different randomly generated trees, edge crossings can not always be eliminated. But our algorithm avoids them



(a) Total Crossings



(b) Final Layout Crossings

Figure 26: Experiment 2 – Spanning-Tree-to-Spanning-Tree Transitions
The results above show the number of crossing produced by the two visualization schemes when transitioning between two different spanning tree drawings.

more successfully than Yee et al.. From our general observations it appears that Yee et al.'s algorithms are prone to produce more crossings when a vertex with few adjacent vertices is chosen as the new root of a spanning tree. Our algorithms did not appear to have this problem when the same vertex was chosen in these certain trials.

7.5.3 Experiment 3 – Full-Graph-to-Spanning-Tree Transitions

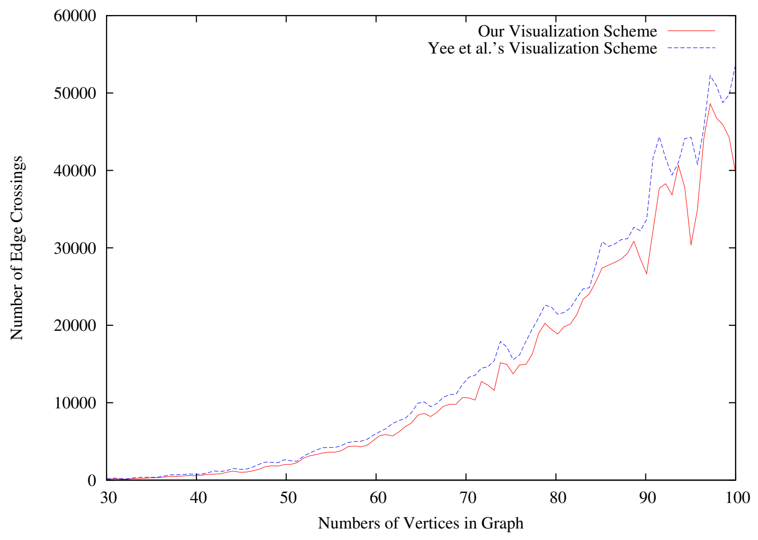
When transitioning to a full graph drawing to spanning-tree-based drawing rooted a randomly selected vertex, our visualization scheme consistently and reliably produces fewer edge crossings. As shown in Figure 27 (page 59), our algorithms produced 40% fewer final layout crossings and 12% fewer overall crossings than Yee et al.'s algorithms over all experiments trials.

7.5.4 Experiment 4 – Spanning Tree Sibling Edge Lengths

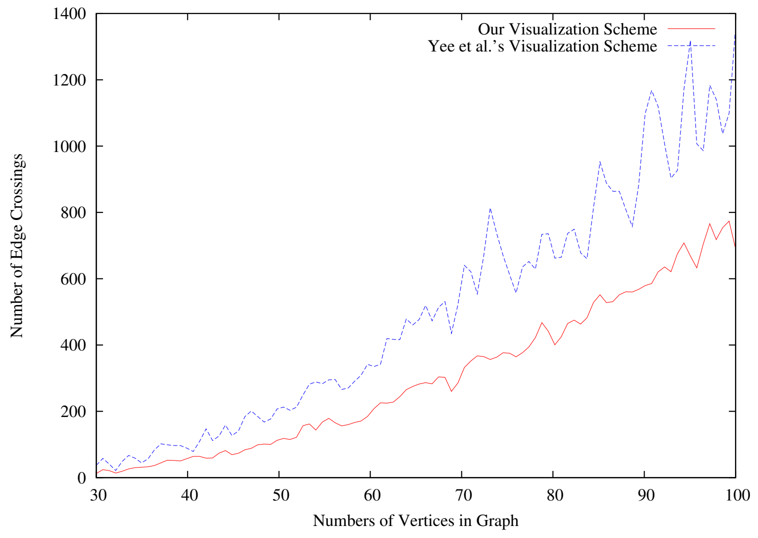
Lastly, we measure the edge lengths of sets of sibling vertices in a graph drawing. We use the final spanning tree drawings generated in experiment 3. For each trial, we calculate the mean length of the edges from child vertices to their parent and then determine the mean standard deviation for all sets of siblings in the graph.

As shown in Figure 28, our graph drawing algorithm creates drawings with no variance in the lengths of edge for sibling vertices to their parent. This is because our algorithm places sibling vertices on circles centered at their parent vertex, and thus are always equally distant from the parent.

One trend worth noting with both drawing algorithms is the decreasing mean edge length in the drawings as the graph order increases. This occurs because increasing vertex connectivity creates shorter path distances from all vertices to the root vertex. We believe this is why drawings generated

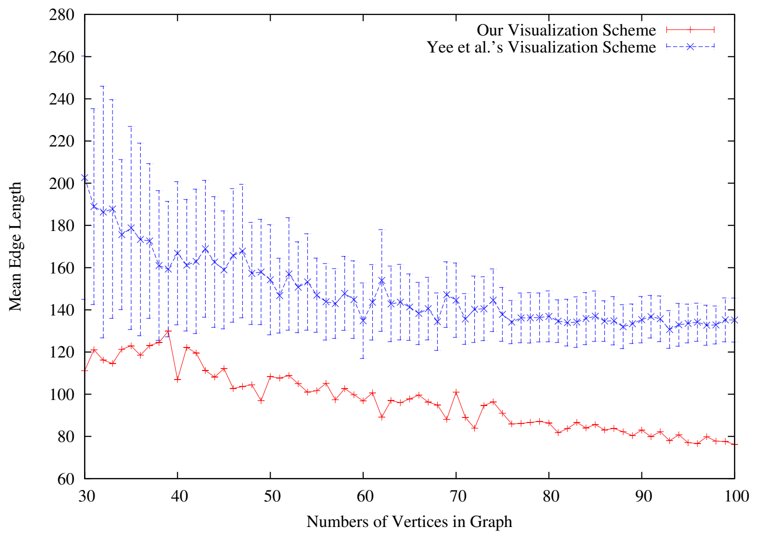


(a) Total Crossings



(b) Final Layout Crossings

Figure 27: Experiment 3 – Full-Graph-to-Spanning-Tree Transitions
The results above show the number of crossing produced by the two visualization schemes when transitioning from a force-directed full graph drawing to a spanning tree drawing.



(a) Mean Length of Sibling Edges with Standard Deviation

Figure 28: Experiment 4 – Spanning Tree Sibling Edge Lengths
The results above show that our visualization scheme produces drawings where all sets of sibling vertices are equidistant to their parent vertex, while Yee et al.'s algorithms fail to generate drawings with this property.

by Yee et al.'s algorithm have such a wide edge length variance for smaller sized graphs; subtrees tend to have greater heights in these smaller graphs and children vertices spread out more on higher concentric circle levels.

8 Discussion & Future Work

Our experiments indicate that drawings and animated transitions generated by our visualization system are significantly simpler than Yee et al.'s Gnutellavision application. The drawings produced by our algorithm make the structural properties of graphs apparent, and conform to many established aesthetics for graph drawings. And in contrast to Yee et al.'s visualization scheme, our transitions produce zero edge crossings in circumstances where zero crossings are necessary. Thus, on objective measures of complexity like crossings and edge lengths, our visualization scheme performs better than Yee et al.'s algorithms. Taken in the context of the theory reviewed in Section 3.1.2, these results suggest that our system should help users make accurate judgments about graph structures. While behavioral experiments would be needed to confirm this hypothesis, we believe that subjects would get less confused and make more reliable judgments about graphs using our methods than Yee et al.'s system. Our research thus lays the ground work for future study of the psychological significance of our metrics and of the functional validity of the graph aesthetics themselves.

Further research is also needed to create a set of established graph animation aesthetics similar to what currently exists for graph drawings [8, 58]. Although research has been conducted to measure usefulness of adding movement to a single graph drawing as means of annotation [10, 68, 69], we are not aware of similar experiments for transitioning between multiple drawings.

With regard to our algorithms, two areas are particularly ripe for further study. First, the drawings produced by our graph drawing algorithm are not guaranteed to be planar; edge crossings may occur when long subtrees encroach on neighboring containment circles. Research into other methods for annulus wedge allocation could lead to an enhanced version of our drawing algorithm that always produces planar drawings.

Second, with our approach remote descendants become vanishingly small on the viewing plane. Our system doe give users a natural solution to this problem (selecting a different focal point vertex so as to allocate more space to the descendants). However, future research could explore the algorithmic relation between this solution versus the similar solution implemented by hyperbolic visualization's distortion of the viewing plane [46, 50]. There are clearly differences: we position and move siblings by constraining them to circles on a Euclidean plane centered at the parent; Hyperbolic layout algorithms position and move siblings through a non-Euclidean space. Future research could also determine whether one of these approaches is more supportive of user-judgments.

9 Conclusion

We have developed an interactive graph visualization system that allows users to explore the structure of a graph through multiple vertex-centric drawings. We introduced a graph drawing algorithm that generates spanning-tree-based drawings for a graph using root vertices selected by the user. In these drawings, vertices are positioned on a series of overlapping circles using their parent vertex as a point of reference. We also introduced a graph animation algorithm that generates smooth, continuous transitions from one graph drawing to another by interpolating vertices' polar coordinates. Transitions created by our algorithm produce no crossings between edges of sibling vertices or between adjacent edges in spanning trees.

We conducted experiments to compare to our experimental system with Yee et al.'s Gnutellavision graph visualization system [73]. Our algorithms were able to transition between two drawings of the same tree with no edge crossings whereas Yee et al.'s system often produced crossings. Our algorithms also transitioned between multiple spanning-tree-based drawings for graphs with fewer edge crossings than Yee et al.'s algorithms. We demonstrated that our algorithms create drawings where sets of sibling vertices are always equidistant to their parent vertex, and measured the degree to which Yee et al.'s drawings did not have this property. These results suggest that our visualization and animation schemes have considerable promise in helping users understand and explore graphs.

References

[1] Alexander Aiken, Jolly Chen, Mark Lin, Mybrid Spalding, Michael Stonebraker, and Allison Woodruff. The tioga-2 database visualization environment. In *Workshop on Database Issues for Data Visualization*, pages 181–207, 1995. URL `citeseer.ist.psu.edu/article/aiken96tioga.html`.

[2] Siew Cheong Au, Christopher Leckie, Ajeet Parhar, and Gerard Wong. Efficient visualization of large routing topologies. *Int. J. Netw. Manag.*, 14(2): 105–118, 2004. ISSN 1099-1190. doi: http://dx.doi.org/10.1002/nem.511.

[3] Huffaker B., Nemeth E., and Claffy K. Otter: a general-purpose network visualization tool. In *Proceedings of the 9th Annual Conference of the Internet Society*, 1999.

[4] Karl-Friedrich B&#246;hringer and Frances Newbery Paulisch. Using constraints to achieve stability in automatic graph layout algorithms. In *CHI '90: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 43–51, New York, NY, USA, 1990. ACM Press. ISBN 0-201-50932-6. doi: http://doi.acm.org/10.1145/97243.97250.

[5] Ricardo A. Baeza-Yates, R. Baeza-Yates, and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 020139829X.

[6] Carlo Batini, L. Furlani, and Enrico Nardelli. What is a good diagram? a pragmatic approach. In *Proceedings of the Fourth International Conference on Object-Oriented and Entity-Relationship Modelling*, pages 312–319, Washington, DC, USA, 1985. IEEE Computer Society. ISBN 0-444-87951-X.

[7] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry: Theory and Applications*, 4(5):235–282, 1994. ISSN 0925-7721. doi: http://dx.doi.org/10.1016/0925-7721(94)00014-X.

[8] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs.* Prentice Hall, Upper Saddle River, New Jersey, 1999.

[9] Richard A. Becker, Stephen G. Eick, and Allan R. Wilks. Visualizing network data. *IEEE Transactions on Visualization and Computer Graphics*, 1(1):16–28, 1995. URL `citeseer.ist.psu.edu/becker95visualizing.html`.

[10] John Bovey, Peter Rodgers, and Florence Benoy. Movement as an Aid to Understanding Graphs. In *Seventh International Conference on Information Visualization (IV03)*, pages 472–478. IEEE, July 2003. ISBN 0-7695-1988-1. URL `http://www.cs.kent.ac.uk/pubs/2003/1653`.

[11] Ulrik Brandes and Dorothea Wagner. A bayesian paradigm for dynamic graph layout. In *GD '97: Proceedings of the 5th International Symposium on Graph Drawing*, pages 236–247, London, UK, 1997. Springer-Verlag. ISBN 3-540-63938-1.

[12] L. M. Burns, J. L. Archibald, and A. Malhotra. A graphical entity-relationship database browser. In *Proceedings of the Twenty-First Annual Hawaii International Conference on Software Track*, pages 694–704, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press. ISBN 0-8186-0842-0.

[13] Yih-Farn (Robin) Chen. Dagger: A tool to generate program graphs. pages 19–35, 1994. URL `citeseer.ist.psu.edu/87704.html`.

[14] Christian Collberg, Stephen Kobourov, Jasvir Nagra, Jacob Pitts, and Kevin Wampler. A system for graph-based visualization of the evolution of software. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 77–ff, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-642-0. doi: http://doi.acm.org/10.1145/774833.774844.

[15] H. S. M. Coxeter. *Non-Euclidean Geometry.* The Mathematical Association of America, 6th edition, September 1998.

[16] Ron Davidson and David Harel. Drawing graphs nicely using simulated annealing. *ACM Trans. Graph.*, 15(4):301–331, 1996. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/234535.234538.

[17] Stephan Diehl and Carsten Gerg. Graphs, they are changing. In *GD '02: Revised Papers from the 10th International Symposium on Graph Drawing*, pages 23–30, London, UK, 2002. Springer-Verlag. ISBN 3-540-00158-1.

[18] Reinhard Diestel. *Graph Theory.* Springer, New York, 1997.

[19] Peter Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42: 149–160, 1984. ISSN 0384-9864.

[20] Peter Eades. Drawing free trees. *Bulletin of the Institute of Combinatorics and its Applications*, 5:10–36, 1992.

[21] Peter Eades and Mao Lin Huang. Navigating clustered graphs using force-directed methods. *J. Graph Algorithms and Applications: Special Issue on Selected Papers from 1998 Symp. Graph Drawing*, 4(3):157–181, 2000. URL `citeseer.ist.psu.edu/eades00navigating.html`.

[22] Peter Eades, W Lai, Kazuo Misue, and Kozo Sugiyama. Preserving the mental map of a diagram. In Harold P. Santo, editor, *Compugraphics '91: First international conference on computational graphics and visualization techniques*, pages 34–43, September 1991.

[23] John Eklund, James Sawers, and Romain Zeiliger. Nestor navigator: A tool for the collaborative construction of knowledge through constructive navigation. In *Proceedings of Ausweb '99 The Fifth Australian World Wide Web Conference*. Southern Cross University Press, 1999.

[24] P. Erdös and A. Rényi. On the evolution of random graphs. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, 5:17–61, 1960.

[25] C. Erten, S. G. Kobourov, and C. Pitta. Morphing planar graphs. In *SCG '04: Proceedings of the twentieth annual symposium on Computational geometry*,

pages 451–452, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-885-7. doi: http://doi.acm.org/10.1145/997817.997886.

[26] Deborah Estrin, Mark Handley, John Heidemann, Steven McCanne, Ya Xu, and Haobo Yu. Network visualization with the VINT network animator nam. Technical Report 99-703b, University of Southern California, March 1999. URL `http://www.isi.edu/ johnh/PAPERS/Estrin99d.html`. revised November 1999, to appear in IEEE Computer.

[27] Arne Frick, Andreas Ludwig, and Heiko Mehldau. A fast adaptive layout algorithm for undirected graphs. In Roberto Tamassia and Ioannis G. Tollis, editors, *Proc. DIMACS Int. Work. Graph Drawing, GD*, number 894, pages 388–403, Berlin, Germany, 10–12 1994. Springer-Verlag. ISBN 3-540-58950-3. URL `citeseer.ist.psu.edu/frick94fast.html`.

[28] Carsten Friedrich. The ffgraph library, 1995. URL `citeseer.ist.psu.edu/friedrich95ffgraph.html`.

[29] Carsten Friedrich and Peter Eades. Graph drawing in motion. *J. Graph Algorithms Appl.*, 6(3):353–370, 2002.

[30] Carsten Friedrich and Michael E. Houle. Graph drawing in motion II. In *GD '01: Revised Papers from the 9th International Symposium on Graph Drawing*, pages 220–231, London, UK, 2002. Springer-Verlag. ISBN 3-540-43309-0.

[31] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, January 1979. ISBN 0716710455.

[32] Ashim Garg and Roberto Tamassia. On the computational complexity of upward and rectilinear planarity testing. *SIAM J. Comput.*, 31(2):601–625, 2001. ISSN 0097-5397. doi: http://dx.doi.org/10.1137/S0097539794277123.

[33] Jeffrey Heer, Stuart K. Card, and James A. Landay. prefuse: a toolkit for interactive information visualization. In *CHI '05: Proceedings of the*

SIGCHI conference on Human factors in computing systems, pages 421–430, New York, NY, USA, 2005. ACM Press. ISBN 1-58113-998-5. doi: http://doi.acm.org/10.1145/1054972.1055031.

[34] Herman, G. Melançon, and M. S. Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, 2000. URL `citeseer.ist.psu.edu/herman00graph.html`.

[35] Hiroshi Hosobe. A high-dimensional approach to interactive graph visualization. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1253–1257, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-812-1. doi: http://doi.acm.org/10.1145/967900.968155.

[36] Mao Lin Huang, Peter Eades, and Robert F. Cohen. Webofdav - navigating and visualizing the web on-line with animated context swapping. In *WWW7: Proceedings of the seventh international conference on World Wide Web 7*, pages 638–642, Amsterdam, The Netherlands, The Netherlands, 1998. Elsevier Science Publishers B. V. doi: http://dx.doi.org/10.1016/S0169-7552(98)00054-3.

[37] Mao Lin Huang, Peter Eades, and Robert F. Cohen. Webofdav navigating and visualizing the web on-line with animated context swapping. In *WWW7: Proceedings of the seventh international conference on World Wide Web 7*, pages 638–642, Amsterdam, The Netherlands, The Netherlands, 1998. Elsevier Science Publishers B. V. doi: http://dx.doi.org/10.1016/S0169-7552(98)00054-3.

[38] Mao Lin Huang, Peter Eades, and Junhu Wang. Online animated graph drawing using a modified spring algorithm. *Australian Computer Science Comm.: Proc. 21st Australasian Computer Science Conf., ACSC*, 20(1):17–28, 4–6 1998. URL `http://citeseer.ist.psu.edu/huang98online.html`.

[39] Weidong Huang and Peter Eades. How people read graphs. In *CRPIT '45: proceedings of the 2005 Asia-Pacific symposium on Information visualisation*,

pages 51–58, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc. ISBN 1-920-68227-9.

[40] Young Hyun. Walrus - a graph visualization tool (http://www.caida.org/tools/visualization/walrus/index.xml), 2002. URL `http://www.caida.org/`.

[41] T. J. Jankun-Kelly and Kwan-Liu Ma. Moiregraphs: Radial focus+context visualization and interaction for graphs with visual nodes. In *Proceedings of the IEEE Symposium on Information Visualization*, volume 00, pages 59–66, October 2003.

[42] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, 1989. ISSN 0020-0190. doi: http://dx.doi.org/10.1016/0020-0190(89)90102-6.

[43] Ernst Kleiberg, Huub van de Wetering, and Jarke J. Van Wijk. Botanical visualization of huge hierarchies. In *INFOVIS '01: Proceedings of the IEEE Symposium on Information Visualization 2001 (INFOVIS'01)*, page 87, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1342-5.

[44] Hideki Koike and Hirotaka Yoshihara. Fractal approaches for visualizing huge hierarchies. In Ephraim P. Glinert and Kai A. Olsen, editors, *Proc. IEEE Symp. Visual Languages, VL*, pages 55–60. IEEE Computer Society, 24–27 1993. ISBN 0-81863-970-9. URL `citeseer.ist.psu.edu/koike93fractal.html`.

[45] S. M. Kosslyn. Understanding charts and graphs. *Applied Cognitive Psychology*, 1(1), 1989.

[46] John Lamping, Ramana Rao, and Peter Pirolli. A focus+context technique based on hyperbolic geometry for visualizing large hierarchies. In *CHI '95: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 401–408, New York, NY, USA, 1995.

ACM Press/Addison-Wesley Publishing Co. ISBN 0-201-84705-1. doi: http://doi.acm.org/10.1145/223904.223956.

[47] Jill H. Larkin and Herbert A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11(1):65–100, 1987.

[48] Guy Melancon and Ivan Herman. Circular drawings of rooted trees. Technical report, Amsterdam, The Netherlands, The Netherlands, 1998.

[49] Kazuo Misue, Peter Eades, Wei Lai, and Kozo Sugiyama. Layout adjustment and the mental map. *J. Vis. Lang. Comput.*, 6(2):183–210, 1995.

[50] Tamara Macushla Munzner. *Interactive visualization of large graphs and networks*. PhD thesis, 2000. Adviser-Pat Hanrahan.

[51] Keith V. Nesbitt and Carsten Friedrich. Applying gestalt principles to animated visualizations of network data. In *IV*, pages 737–743, 2002.

[52] Quang Vinh Nguyen and Mao Lin Huang. A space-optimized tree visualization. In *INFOVIS '02: Proceedings of the IEEE Symposium on Information Visualization (InfoVis'02)*, page 85, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1751-X.

[53] Stephen C. North. Incremental layout in dynadag. In *GD '95: Proceedings of the Symposium on Graph Drawing*, pages 409–418, London, UK, 1996. Springer-Verlag. ISBN 3-540-60723-4.

[54] Achilleas Papakostas and Ioannis G. Tollis. Interactive orthogonal graph drawing. *IEEE Trans. Comput.*, 47(11):1297–1309, 1998. ISSN 0018-9340. doi: http://dx.doi.org/10.1109/12.736444.

[55] T. Pavlidis. Linear and context-free graph grammars. *J. ACM*, 19(1):11–22, 1972. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321679.321682.

[56] Mark Phillips and Charlie Gunn. Visualizing hyperbolic space: unusual uses of 4x4 matrices. In *SI3D '92: Proceedings of the 1992 symposium on Interactive*

3D graphics, pages 209–214, New York, NY, USA, 1992. ACM Press. ISBN 0-89791-467-8. doi: http://doi.acm.org/10.1145/147156.147206.

[57] Andrzej Proskurowski. Report on gra-gra: 4th international workshop on graph grammars and their applications to computer science. *SIGACT News*, 21(2):39, 1990. ISSN 0163-5700. doi: http://doi.acm.org/10.1145/379172.379187.

[58] Helen C. Purchase. The effects of graph layout. In *OZCHI '98: Proceedings of the Australasian Conference on Computer Human Interaction*, page 80. IEEE Computer Society, 1998. ISBN 0-8186-9206-5.

[59] R.C. Read. Methods for computer display and manipulation of graphs and the corresponding algorithms. Technical report, Faculty of Mathematics, University of Waterloo, July 1986.

[60] Gruia-Catalin Roman and Kenneth C. Cox. Program visualization: The art of mapping programs to pictures. In *International Conference on Software Engineering*, pages 412–420, 1992. URL `citeseer.ist.psu.edu/roman92program.html`.

[61] Manojit Sarkar and Marc H. Brown. Graphical fisheye views of graphs. In *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 83–91, New York, NY, USA, 1992. ACM Press. ISBN 0-89791-513-5. doi: http://doi.acm.org/10.1145/142750.142763.

[62] Doug Schaffer, Zhengping Zuo, Saul Greenberg, Lyn Bartram, John Dill, Shelli Dubs, and Mark Roseman. Navigating hierarchically clustered networks through fisheye and full-zoom methods. *ACM Trans. Comput.-Hum. Interact.*, 3(2):162–188, 1996. ISSN 1073-0516. doi: http://doi.acm.org/10.1145/230562.230577.

[63] Ben Shneiderman. *Designing the user interface: strategies for effective human-computer interaction.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-16505-8.

[64] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical systems. *IEEE Transactions on Systems, Man, an Cybernetics.*, SMC-11(2):109–125, 1981.

[65] Roberto Tamassia, Giuseppe Di Battista, and Carlo Batini. Automatic graph drawing and readability of diagrams. *IEEE Trans. Syst. Man Cybern.*, 18(1): 61–79, 1988. ISSN 0018-9472. doi: http://dx.doi.org/10.1109/21.87055.

[66] Soon Tee Teoh and Kwan-Liu Ma. Rings: A technique for visualizing large hierarchies. In *GD '02: Revised Papers from the 10th International Symposium on Graph Drawing*, pages 268–275, London, UK, 2002. Springer-Verlag. ISBN 3-540-00158-1.

[67] Edward Tufte. *The Visual Display of Quantitative Information.* Graphics Press, Cheshire CT, 1983. ISBN 0-9613921-0-X.

[68] Colin Ware and Robert Bobrow. Motion to support rapid interactive queries on node–link diagrams. *ACM Trans. Appl. Percept.*, 1(1):3–18, 2004. ISSN 1544-3558. doi: http://doi.acm.org/10.1145/1008722.1008724.

[69] Colin Ware and Glenn Franck. Evaluating stereo and motion cues for visualizing information nets in three dimensions. *ACM Trans. Graph.*, 15(2):121–140, 1996. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/234972.234975.

[70] Colin Ware, Helen Purchase, Linda Colpoys, and Matthew McGill. Cognitive measurements of graph aesthetics. *Information Visualization*, 1(2):103–110, 2002. ISSN 1473-8716. doi: http://dx.doi.org/10.1057/palgrave.ivs.9500013.

[71] Richard John Webber. *Finding the Best Viewpoint for Three-Dimensional Graph Drawings.* PhD thesis, The University of Newcastle, Australia, July 1998.

[72] Graham J. Wills. NicheWorks — interactive visualization of very large graphs. *Journal of Computational and Graphical Statistics*, 8(2):190–212, 1999. URL `citeseer.ist.psu.edu/wills97nicheworksinteractive.html`.

[73] Ka-Ping Yee, Danyel Fisher, Rachna Dhamija, and Marti A. Hearst. Animated exploration of dynamic graphs with radial layout. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 43–50, 2001. URL `citeseer.ist.psu.edu/article/yee01animated.html`.