Visualization Techniques for Probability-Weighted Tree and Automata-Based Models

Victor De Cerqueira cerqueir@cs.utk.edu
Geoff Mazeroff mazeroff@cs.utk.edu
Jens Gregor jgregor@cs.utk.edu
Michael G. Thomason thomason@cs.utk.edu

Department of Computer Science
University of Tennessee
1122 Volunteer Blvd., Suite 203
Knoxville, TN 37996-3450

ABSTRACT

This paper describes a number of visualization techniques for rendering variable-order probabilistic tree and automata models. The techniques, which are presented in the context of a graphical user interface for analysis of malicious mobile code, are approached at three levels: graph layout, user-interaction, and display. At the graph layout level, we describe and compare two popular algorithms: Graphplace, which is suitable for tree drawing due to its simple heuristics for computing edge crossing reductions fast, and Dot, which is better for automata layouts due to its built-in network simplex approach to optimization as well as its spline-based edge routing mechanism. At the user-interaction level, we provide a fast scrolling technique based on a kd-tree search algorithm. Finally, at the display level, we show a number of ways in which the display of a probability-weighted tree or automaton can be fine-tuned in accordance with a variety of properties such as density, edge weight, and other relevant features.

1. INTRODUCTION

In numerous applications, probability-weighted trees and automata inferred from samples of sequences are highly effective models. Real data often yield large, complex models, the understanding of which is aided by effective visualization. This means that wherever possible, the visualization should preserve properties such as symmetry, planarity and node distance, so that subtle structural patterns in the model can be explored in a subsequent data-mining process. This paper describes a number of visualization techniques for rendering variable-order tree and automata models. The techniques are presented in the context of a graphical user interface for analysis of malicious mobile code called SPARTA, which is short for Stochastic Profiling Application for the Rendering of Trees and Automata. Suffice it here to note that SPARTA builds Probabilistic Suffix Trees (PSTs)[2] and Probabilistic Suffix Automata (PSAs)[7] to facilitate the identification of common behavioral patterns among applications, so that a standard of *normal* behavior can be specified and used to flag abnormal (and potentially malicious) behavior in a computer system. We discuss tree and automata visualization at three levels: graph layout, user-interaction, and display. At the graph layout level, we briefly describe some of the paradigms and common approaches to graph drawing and outline the steps of a generic algorithm suitable for both trees and automata. We then discuss the specifics of tree and automata drawing, presenting two popular variations of the generic algorithm, namely Graphplace, which is suitable for tree drawing due to its simple heuristics for computing edge crossing reductions fast, and Dot, which is better for automata layout due to its built-in network simplex approach to optimization as well as its spline-based edge routing mechanism. At the user-interaction level, we provide a fast scrolling technique by introducing a kd-tree search algorithm that allows updating the screen only for visible nodes and edges. Finally, at the display level, we show a number of ways in which the display of a probability-weighted tree or automaton can be fine-tuned in accordance with a variety of properties such as density (which we emphasize by shrinking nodes and drawing them closer together), edge weight (which we use to render unwanted edges invisible as well as to color-code edges that are regarded as having similar properties), and other relevant features. Throughout this paper, trees and automata will be generically referred to as "graphs" whenever properties pertinent to both models are described.

2. THE GRAPH LAYOUT ALGORITHM

Over the past couple of decades, there have been tremendous advances in graph layout techniques. Although the work is still far from being completed (mostly due to the NP-completeness associated with certain steps of current graph layout algorithms), researchers have managed to establish paradigms that are essential for the "successful" drawing of a graph. In this context, success is measured by the degree to which a graph manages to represent a certain model and the dependencies/relationships among its variables with accuracy and intuitiveness. For example, PERT diagrams and subroutine-call graphs are best represented by a hierarchical approach, whereas data flow diagrams are usually represented by orthogonal drawings in the topology-shape-metrics approach. The reader is referred to [1] for further details on these and other popular approaches. For the purposes of SPARTA, the two most important aspects of graph drawing are:

- Aesthetics: The aesthetic goal in graph drawing is to minimize edge crossings, edge bends (so that edges look as straight as possible), total edge length, and total area encompassed by the graph, while maximizing the display of symmetries;

- Computational efficiency: Since SPARTA is intended to be a highly interactive tool, it must provide real-time response to user events. Therefore, even the largest drawings must be laid out in a timely fashion.

It is important to notice that the aesthetic criteria mentioned above often conflict with each other, making it next to algorithmically infeasible to enforce them all at the same time. Tradeoffs are thus unavoidable, and the final layout must rely on heuristics to obtain the best compromise among all the aesthetic criteria adopted and still generate an acceptable visualization of a graph.

SPARTA relies on two independent algorithms to generate layouts for its graphs. PSTs are constructed using a Java port of Graphplace, originally developed by Jos van Eijndhoven at the Eindhoven University of Technology, whereas PSAs are constructed with the sophisticated Dot algorithm, developed in C by Gansner, Koutsofios, North, and Vo at the AT&T Bell Labs[4]. Since trees and automata are basically modeling dependency relationships, the hierarchical approach is adopted in SPARTA. The standard algorithm based on this approach takes as its input a graph description (formally, a set V of vertices and a set E of edges), and generates coordinates for its vertices and edges in four steps:

1. **Cycle Removal**: This is an optional step that takes place whenever the graph contains cycles. Since it is often aesthetically pleasing to generate graphs where most edges flow in the same direction (e.g., from top to bottom), this step transforms a cyclic graph into an acyclic one by reversing some of its edges. The graph is then laid out according to the next three steps, and all reversed edges are restored to their original directions before the algorithm returns;

2. **Layering**: This step transforms the acyclic input graph into a proper layered digraph, where its vertices are assigned to layers $L_1, L_2, ...L_n$ such that the bounding boxes of vertices sitting on the same layer have the same top y-coordinate, and edges never span more than one layer. If the endpoints of an edge are more than one layer apart, then dummy vertices (also known as virtual vertices) are inserted across the layers to enforce the one-layer-span rule. Some of the aesthetic concerns involved in this step are the final width and height of the graph (measured by the size of its bounding box), and the number of dummy vertices created, since it can be quadratic if there are $O(n)$ edges spanning $O(n)$ layers;

3. **Crossing Reduction**: As the name suggests, this step receives as an input the layered digraph, and reorders the vertices on each layer in such a way that the number of edges crossing is minimum. Unfortunately, the problem of reducing edge crossings is equivalent to the combinatorial problem of choosing appropriate vertex orderings for each layer, thus being NP-complete[5]. This problem can be dealt with in a variety of ways, all of them applying some sort of heuristics; the most common approaches are the Barycenter and Median methods[1], since they run in linear time and are capable of generating planar layouts (i.e., layouts where no two edges cross) whenever one exists for a given input graph;

4. **Horizontal Coordinate Assignment**: This step assigns x-coordinates to the vertices in each layer, respecting the ordering computed in the crossing reduction step. Further aesthetic refinement in the drawing can be achieved in this step by vertically aligning dummy vertices, since this reduces the number of bends in long edges. Although conceptually simple, this step relies on quadratic programming techniques[4] that require a great deal of computational effort.

The next two sections describe how this generic algorithm is used in the visualization of trees and automata.

3. TREE DRAWING

Because of their planar representation, trees can usually be laid out much more easily than a generic graph. For this reason, Graphplace is used to render PSTs in SPARTA. This algorithm offers a simple, yet effective implementation of the graph layout algorithm described in the previous section. It bestows a very fast linear-time heuristic for node placement, traversing the graph in a Top-Down (North-South), depth-first search order along the edges, which are just assumed to be directed (this has no implication in the layout of undirected graphs, which is the case of PSTs). Furthermore, the Java port of Graphplace used in SPARTA completely ignores the cycle removal step, since trees are by definition acyclic undirected graphs. The overall computational complexity of the algorithm is $O(n \log n)$, where n is the number of vertices, which allows fast placement of even very large graphs. The limitations in the placement provided by Graphplace rest in the following facts:

- Nodes are assumed to be represented as dots (i.e., they have no area). This can cause nodes to overlap if their areas cover the vertical / horizontal space that separates them. In SPARTA, where PST nodes are represented by ellipses, this problem was fixed by specifying maximum values for the widths and heights of the nodes' bounding boxes and making the horizontal and vertical distances between any two nodes greater than those values;

- The layout technique implicitly assumes that edges are drawn as polylines containing the dummy nodes created in the layering step as vertices in a polygonal chain. This is not a problem in tree drawing, since tree edges can always be drawn as straight lines without the aid of dummy nodes (see Fig. 1). However, the display of a more generic graph such as an automaton would be impaired by a polyline approach, as it would be more desirable and intuitive to draw the edges as spline segments.

These limitations are next to harmless in tree drawing, and in fact help the Graphplace algorithm run faster. They do, however, show why Graphplace would not be as effective in the layout of more general graphs. For one thing, the assumption that nodes are represented as points could not be handled so easily, since overlaps might occur not only between two ordinary nodes, but also between nodes and dummy nodes, which would result in edges running over nodes other than its endpoints or control points. If on the one hand dimension constraints could be applied to the nodes' bounding boxes to avoid node-node overlap, applying the same constraints to dummy nodes would be likely to result in unnecessarily large layouts that would fail to wisely explore the spaces between nodes in the routing of edges. Furthermore, the simple heuristics used in the cross reduction and x-coordinate assignment steps of Graphplace usually are not very effective in the layout

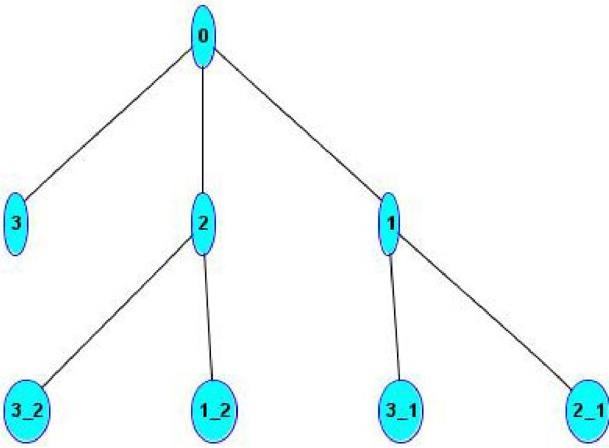

Figure 1: A small PST.

of more generic graphcs, resulting in excessive edge crossing and sharp bends of long edges.

Since nodes in a tree are by construction one layer apart from their parents, straight lines can be used to connect a node and its offspring. In SPARTA, this is accomplished by defining the endpoints of an edge to be the center of the parent's bounding box, and the nearest point on the children's elliptical shape. Let P denote the parent node's center point, and P' its nearest point on the child's ellipse. $P = (x_c, y_c)$ can be easily calculated by the following formulas:

$$x_c = x + w/2, \qquad y_c = y + h/2,$$

where x and y are the left and top coordinates, and w and h are the width and height of the parent node's bounding box. Obtaining P' is more problematic, since its calculation depends on equations of polynomials of degree 4, for which there is no closed-form solution[3]. A numerical approximation could be attempted, but this could slow down the rendering process significantly. Since geometric accuracy is not absolutely essential in the context of PST rendering (the only hard requirement is that P' lies on the child's ellipse), it is possible to use an altered version of the formula for the nearest point on a circle to generate acceptable results. This formula for $P' = (x_n, y_n)$ is given below:

$$x_n = x'_c + \frac{(w'/2)*(x_c - x'_c)}{\sqrt{(x_c - x'_c)^2 + (y_c - y'_c)^2}}, y_n = y'_c + \frac{(h'/2)*(y_c - y'_c)}{\sqrt{(x_c - x'_c)^2 + (y_c - y'_c)^2}},$$

where x'_c, y'_c, w', and h' are the child node's analogous values to x_c, y_c, w, and h.

4. AUTOMATA DRAWING

The need for a more powerful layout algorithm for drawing PSAs led to the use of Dot. This algorithm offers improvements on all the major issues with Graphplace aforementioned:

- Node dimensions are considered when generating coordinates for their bounding boxes. These dimensions can be custom-specified, and widths can even be automatically calculated based on the label lengths of the nodes;
- A more complex version of the Median method is used for cross reduction, resulting in "cleaner" drawings and more straight edges. This is very useful when displaying large PSAs with numerous nodes and edges;
- A powerful spline routing mechanism which transforms the dummy points of the layering step into B-spline control points is used. This results in very smooth splines, reducing a great deal the "spaghetti effect" caused by unnecessary bends in an edge (see Fig. 2). The spline routing mechanism also supports self-edges (or "loops") in the layout, which are quite common in automata drawing.

Dot is also innovative in that it relies on a network simplex algorithm (NSA) during the steps that require heuristics. The method is a variation of the popular Linear Programming algorithm known as simplex, which basically tries to find an optimal solution to a problem by iteratively generating intermediate (and increasingly better) solutions until an optimality criterion is met. The network version of the algorithm tries to make the number of iterations smaller by approaching the problem in a geometrical, graphical context, rather than as a numerical one. The NSA is used in two occasions in the layout process. First, to solve a linear programming problem (which can be translated into an integer programming problem in polynomial time) of generating a "ranking" (i.e., a certain vertical ordering of nodes analogous to the layering step of the basic algorithm aforementioned) of nodes that results in the shortest edges (whose lengths are associated with weights which have to be greater than a certain threshold, usually one). The NSA is used again in the assignment of x-coordinates by solving the problem of finding an arrangement of nodes along the x-axis where edges between nodes in adjacent ranks remain as close to vertical as possible. In other words, the NSA tries to assign close x values to the endpoints of a given edge. These two instances of the algorithm (especially the second one) are used because they are easy to program, but as a drawback they can be rather costly for medium to large graphs, since the size of the simplex matrix can grow from VE to $2VE + E^2$ entries (where V is the number of vertices, and E the number of edges).

In order to take advantage of the spline routing mechanism of Dot and generate the smoothest possible layout for automata, SPARTA uses Java's $CubicCurve$ object[6] to interpolate splines along the dummy nodes created by Dot. Nevertheless, before these points can be used, they must first be converted into Bezier control points, since that is the only geometry supported by Java. The conversion is straightforward, and uses the Bezier and B-spline characteristic matrices (denoted by B_z and B_s, respectively) shown below:

$$B_z = \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

$$B_s = \frac{1}{6} \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{pmatrix}$$

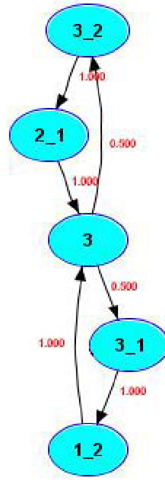

Figure 2: A small PSA.

Define $P(i)$ to be a B-spline control point, with i ranging from 1 to n-3, where n is the number of control points. Then,

$$P = \begin{bmatrix} P(i-1) \\ P(i) \\ P(i+1) \\ P(i+2) \end{bmatrix}$$

can be converted into P' in Bezier geometry through the following formula:

$$P' = B_z^{-1} B_s P$$

Another issue involved in the drawing of PSA edges is the fact that automata are directed graphs, and hence one of the endpoints of an edge must be combined with an arrowhead. Since Java does not provide an arrowhead object, we approached the problem geometrically. Specifically, arrowheads are represented as isosceles triangles whose heights are collinear with the line connecting two points, namely A(the closest control point to the child's ellipse) and B(the closest point on the child's ellipse). Let the subscripts x and y denote the horizontal and vertical coordinates for a given point, and R be a constant positive integer value. Then, the vertices P_0, P_1, and P_2 of the triangle are obtained through the following formulas:

$$P_0 = (B_x - D_x R + D_y \frac{R}{2}, B_y - D_y R - D_x \frac{R}{2})$$

$$P_1 = (B_x, B_y)$$

$$P_2 = (B_x - D_x R - D_y \frac{R}{2}, B_y - D_y R + D_x \frac{R}{2})$$

where

$$D_x = \frac{B_x - A_x}{\sqrt{(B_x - A_x)^2 + (B_y - A_y)^2}}, D_y = \frac{B_y - A_y}{\sqrt{(B_x - A_x)^2 + (B_y - A_y)^2}}.$$

Without loss of generality, assume the arrowhead is pointing upwards (otherwise rotate its vertices so that it points upwards). Then, P_0 is the left base vertex, P_1 is the top vertex, and P_2 is the right base vertex. The arrowhead is rendered by connecting the points P_0, P_1, and P_2.

5. SCROLLING ACROSS LARGE GRAPHS

The models considered can yield fairly large and complex graphical representations. This makes it infeasible to fit the entire graph on the screen. One possible remedy is to scale the layout according to the size of the graph, but this can be impractical because it can cause nodes to be rendered too small, making it difficult to read their labels. Another plausible solution is to implement zooming so that particular regions of a graph could be observed in more detail, but this would still be ineffective because of the possible slowdown caused by re-rendering the graph every time a zoom action was performed, which would be somewhat of a nuisance for the user. A better alternative is to implement scrolling so that a graph can be rendered in its full size and according to the aesthetic criteria outlined above. The resulting display is a compromise between seeing a good portion of the graph on the screen and easy visualization of node labels.

SPARTA implements graph scrolling by means of a class called $VertexWindow$, which is used to maintain a record of nodes that are visible on the screen given the position of the $viewport$.[1] The underlying data structure used by the $VertexWindow$ class is a kd-tree, which is considered the best among "several exotic structures that support range searching"[8]. The kd-tree instance used in SPARTA is a two-dimensional binary search tree (or 2d-tree), which operates by returning nodes that fall within two ranges at the same time: one for values along the x-axis, and one for values along the y-axis. These ranges are denoted by [*top,top+height*] and [*left,left+width*], where *top* and *left* are the coordinates for the viewport's upper left corner, and *width* and *height* are the viewport's width and height. These values are updated whenever a scrolling or resizing action occurs, and are immediately fed to the $VertexWindow$ object, which performs a range query in its 2d-tree and returns the visible nodes in $O(m + \sqrt{n})$ (where m is the number of matches) time, since the 2d-tree is always a perfectly balanced tree. This is possible because once a graph is laid out, nodes are never added to or removed from it.

6. VISUAL ENHANCEMENTS

Having described more general visualization techniques in the previous sections, we now present specific ways to fine-tune the display of graphs by means of four independent techniques which have proven to be useful in the data-mining process of profiling application behavior. These techniques, which can be applied to any type of visual data modeling based on trees and automata, are as follows:

- **Skeleton Mode**: The large, complex graph layouts yielded by real data might obfuscate relevant structural patterns in the final display, jeopardizing the data-mining proccess. In malicious mobile code profiling, it is imperative that such patterns do not go unnoticed, and SPARTA deals with this by offering an alternative visualization of its trees and automata, in what we call $skeleton$ mode. This visualization emphasizes the density of a graph over individual node information by shrinking the nodes of the graph and drawing them closer together so that more nodes fit in the viewport. This can be seen as a rudimentary, yet practical implementation of zooming, in that it is easy to code and allows the user to quickly switch between visualization modes without having to specify resize factors. This sort of structural visualization

[1] A $viewport$ is a Java object denoting the visible part of a panel when the latter is larger than its bounding frame. The viewport moves upon a scrolling action, revealing previously hidden (and hiding previously visible) areas of the panel.

is especially useful for trees, since their planar layouts offer clean displays where patterns might be observed more easily (see Fig. 3).

- **Edge Color-Coding**: The traditional technique of labeling an edge with its weight value does not fully explore the potential of a visual representation of a graph and becomes next to useless when the graph is large. Ideally, weight-related patterns should be observed with a simple glance at the graph, and this can be accomplished in a variety of ways. One of them is to make edges thicker as their weights increase, but this might still not be very indicative of patterns if the weights do not vary much. A better way is to color-code edges according to ranges of weight values. In SPARTA, for example, edges have associated probabilities, and the user is allowed to map probability ranges to colors and "paint" edges accordingly (see Fig. 4), or even "turn off" edges that have low probabilities by rendering them invisible. This results in cleaner layouts, where only relevant information is displayed, thus being an important data-mining aid. However, color-coding has to be used with judiciousness to avoid a "rainbow" effect where the excess of colors becomes more distracting than informative.

- **Node Condensing**: Sometimes it is possible to identify node regions that are uninteresting and only occupy space in the display. In SPARTA, this occurs in automata where nodes are chained by edges of probability 1. The user is given the option to condense these nodes inside "supernodes" and still be able to retrieve information pertaining to a particular internal node by selelecting a supernode (see Fig. 4). In many instances this condensing technique reduces the size of automata significantly, speeding up the layout process and offering cleaner and more concise visualizations.

- **Node Blurring**: In SPARTA trees can be merged two at a time, which is representative of combining the sequences of system calls performed by the applications they are modeling. Merged trees maintain a "merging history" in the sense that they provide information on the origin of their nodes, and SPARTA conveys this information by allowing the user to blur out nodes that do not originate from one tree or another (see Fig. 5). More generally, this idea of node blurring can be thought of as a way to distinguish nodes according to specified properties that they have in a similar fashion to what coloring does to edges. This can be very useful in the exploration of composite models (i.e., models assembled by the combination of other models) in that it gives the user the possibility to observe how the different components affect the final model.

7. CONCLUSION

The goal of this paper was to present a number of techniques for generating effective visualizations of probability-weighted tree and automaton models. We have presented implementations of two known algorithms for graph layout, one tailored for tree drawing and one for automaton drawing, and introduced a kd-tree-based scrolling technique that has proven to be fast for even the largest graphs. We have also described particular ways in which the display of a graph can be enhanced and fine-tuned to suit different aspects of a data-mining proccess. We believe that, while the discussion was presented in the context of an application for malicious mobile code investigation, the techniques described in this paper are universal enough to be easily modified to suit any kind of

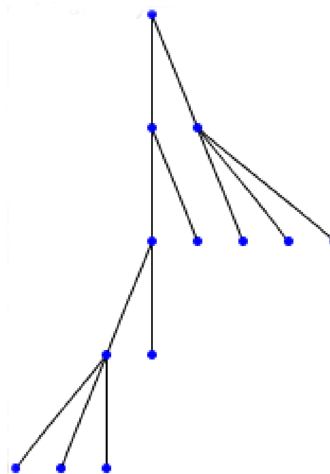

Figure 3: A PST in $skeleton$ mode.

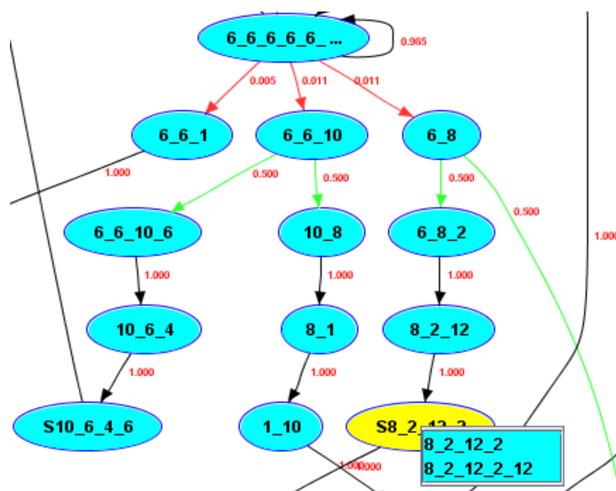

Figure 4: A condensed PSA with color-coded edges. Supernodes' labels are prefixed by an S.

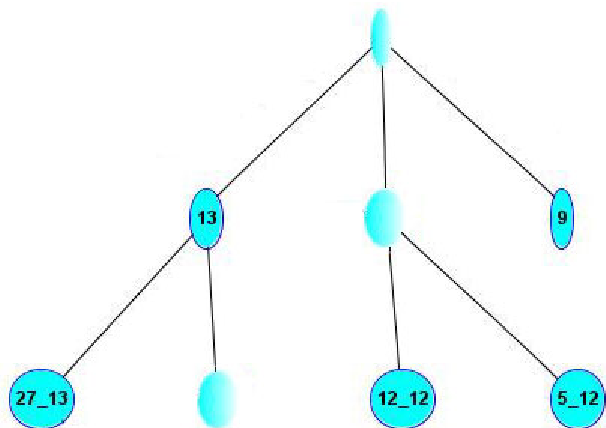

Figure 5: A PST with blurred nodes.

model that can be represented by some type of graph, regardless of it being a tree or automaton.

8. ACKNOWLEDGEMENTS

The work is funded by the Office of Naval Research under grant number N00014-01-1-0862.

9. REFERENCES

[1] G.Di Battista, R.Tamassia P. Eades, and I.G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, Upper Saddle River, NJ, 1998.

[2] G. Bejerano and G. Yona. Variations on probabilistic suffix trees: Statistical modeling and prediction of protein families. *Bioinformatics*, 17(1):23–43, January 2001.

[3] Jr. D.R. Olsen. *Developing User Interfaces*. Morgan Kaufmann, San Francisco, CA, 1998.

[4] E.R. Gansner, E. Koutsofios, S.C. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, March 1993.

[5] M.R. Garey and D.S. Johnson. *Computers and Intractability*. W.H. Freeman and Company, New York, NY, 2002.

[6] V.J. Hardy. *Java 2D API Graphics*. Sun Microsystems Press, Palo Alto,CA, 2000.

[7] D. Ron, Y. Singer, and N. Tishby. The power of amnesia: Learning probabilistic automata with variable memory length. *Machine Learning*, 25(2-3):117–149, 1996.

[8] M.A. Weiss. *Data Structures & Algorithm Analysis in C++*. Addison-Wesley, New York, NY, 1999.