

CHAPTER 8

Visualization Techniques for Trees, Graphs, and Networks

While most of the visualization techniques discussed thus far focus on the display of data values and their attributes, another important application of visualization is the conveying of relational information, e.g., how data items or records are related to each other. These interrelationships can take many forms:

- part/subpart, parent/child, or other hierarchical relation;
- connectedness, such as cities connected by roads or computers connected by networks;
- derived from, as in a sequence of steps or stages;
- shared classification;
- similarities in values;
- similarities in attributes (e.g., spatial, temporal).

Relationships can be simple or complex: unidirectional or bi-directional, nonweighted or weighted, certain or uncertain. Indeed, the relationships may provide more and richer information than that contained in the data records. Applications for visualizing relational information are equally diverse, from categorizing biological species, to exploring document archives, to studying a terrorist network.

In this chapter we will examine a number of techniques that have been developed for visualizing relational information. This presentation, however, will just be the tip of the iceberg, as tree and graph visualization is a well-established field, with its own books, journals, conferences, software packages, and algorithms.

8.1 Displaying Hierarchical Structures

Trees or hierarchies (we'll use the terms interchangeably) are one of the most common structures to hold relational information. For this reason, many visualization techniques have been developed for display of such information. We can divide these techniques into two classes of algorithms: space-filling and non-space-filling. The rest of this section will provide details on implementing algorithms for visualizing this type of data.

8.1.1 Space-Filling Methods

As the name implies, space-filling techniques make maximal use of the display space. This is accomplished by using juxtapositioning to imply relations, as opposed to, for example, conveying relations with edges joining data objects. The two most common approaches to generating space-filling hierarchies are rectangular and radial layouts.

Treemaps [176] and their many variants are the most popular form of rectangular space-filling layout. In the basic treemap, a rectangle is recursively divided into slices, alternating horizontal and vertical slicing, based on the populations of the subtrees at a given level. Pseudocode for this process is given in Figure 8.1, and an example is shown in Figure 8.2.

As mentioned, many variants on treemaps have been proposed and developed since they were introduced, including *squarified treemaps* [40] (to reduce the occurrence of long, thin rectangles) and nested *treemaps* [176] (to emphasize the hierarchical structure).

The methods described above are structured using horizontal and vertical divisions to convey the hierarchy. A number of other approaches are possible, however, such as those that divide space radially. Radial space-filling *hierarchy visualizations*, sometimes referred to as sunburst displays [336], have the root of the hierarchy in the center of the display and use nested rings to convey the layers of the hierarchy. Each ring is divided based on the number of nodes at that level. These techniques follow a similar strategy to treemaps, in that the number of terminal nodes in a subtree determines

```

Start: Main Program
    Width = width of rectangle
    Height = height of rectangle
    Node = root node of the tree
    Origin = position of rectangle, e.g., [0,0]
    Orientation = direction of cuts, alternating between horizontal and vertical
    Treemap(Node, Orientation, Origin, Width, Height)
End: Main Program

Treemap(node n, orientation o, position orig, hsize w, vsize h)
    if n is a terminal node (i.e., it has no children)
        draw-rectangle(orig, w, h)
        return
    for each child of n (child_i), get number of terminal nodes in subtree
    sum up number of terminal nodes
    compute percentage of terminal nodes in n from each subtree (percent-i)
    if orientation is horizontal
        for each subtree
            compute offset of origin based on origin and width (offset-i)
            treemap(child_i, vertical, orig + offset-i, w * percent-i, h)
    else
        for each subtree
            compute offset of origin based on origin and height (offset-i)
            treemap(child_i, horizontal, orig + offset-i, w, h * percent-i)
    End: Treemap

```

Figure 8.1. Pseudocode for drawing a hierarchy using a treemap.

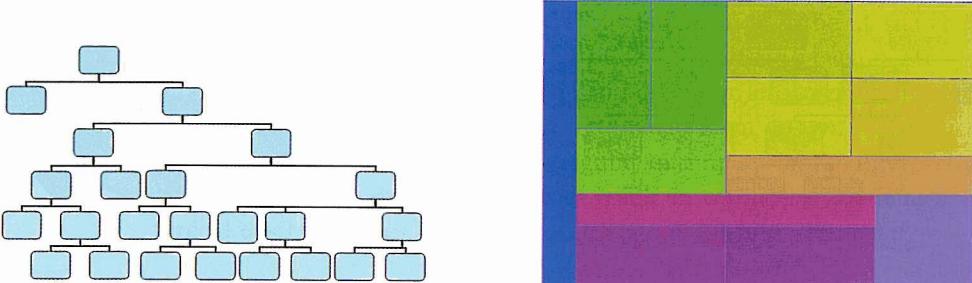


Figure 8.2. A sample hierarchy and the corresponding treemap display.

```

Start: Main Program
  Start = start angle for a node (initially 0)
  End = end angle for a node (initially 360)
  Origin = position of center of sunburst, e.g., [0,0]
  Level = current level of hierarchy (initially 0)
  Width = thickness of each radial band - based on max depth and display size
  Sunburst(Node, Start, End, Level)
End: Main Program

Sunburst(node n, angle st, angle en, level l)
  if n is a terminal node (i.e., it has no children)
    draw_radial_section(Origin, st, en, l * Width, (l+1) * Width)
    return
  for each child of n (child-i), get number of terminal nodes in subtree
  sum up number of terminal nodes
  compute percentage of terminal nodes in n from each subtree (percent_i)
  for each subtree
    compute start/end angle based on size of subtrees, order, and angle range
    Sunburst(child-i, st_i, en_i, l+1)

End: Sunburst

```

Figure 8.3. Pseudocode for drawing a hierarchy using a sunburst display.

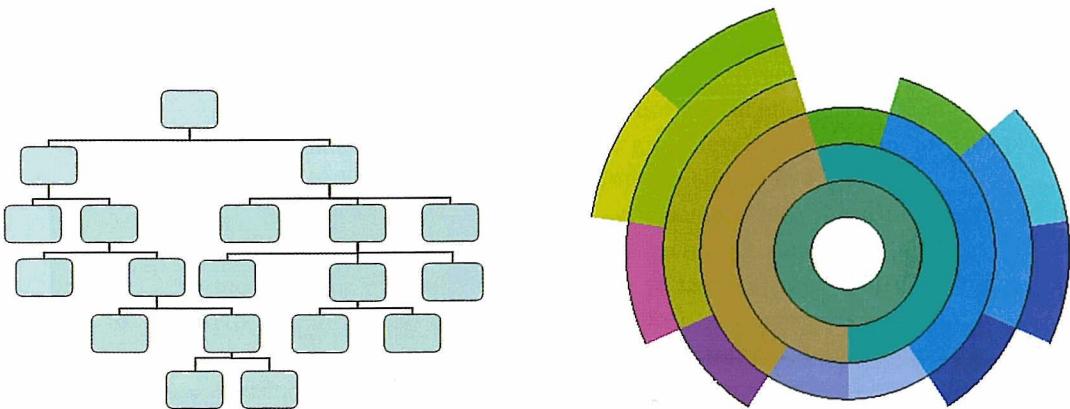


Figure 8.4. A sample hierarchy and the corresponding sunburst display.

the amount of screen space that will be allocated for it. However, unlike treemaps, which assign most screen space to conveying the terminal nodes, radial techniques also show the intermediate nodes. The process is described in pseudocode in Figure 8.3, and an example is shown in Figure 8.4.

For these and other space-filling techniques, color can be used to convey many attributes, such as a value associated with the node (e.g., classification) or it may reinforce the hierarchical relationships, e.g., siblings and parents may have similarities in color, as seen in Figure 8.4. Symbols and other markings may also be embedded in the rectangular or circular segments to communicate other data features.

8.1.2 Non-Space-Filling Methods

The most common representation used to visualize tree or hierarchical relationships is a *node-link diagram*. Organizational charts, family trees, and tournament pairings are just some of the common applications for such diagrams. The drawing of such trees is influenced the most by two factors: the fan-out degree (e.g., the number of siblings a parent node can have) and the depth (e.g., the furthest node from the root). Trees that are significantly constrained in one or both of these aspects, such as a binary tree or a tree with only three or four levels, tend to be much easier to draw than those with fewer constraints.

When designing an algorithm for drawing any node-link diagram (not just trees), one must consider three categories of often-contradictory guidelines: drawing conventions, constraints, and aesthetics. Conventions may include restricting edges to be either a single straight line, a series of rectilinear lines, polygonal lines, or curves. Other conventions might be to place nodes on a fixed grid, or to have all sibling nodes share the same vertical position. Constraints may include requiring a particular node to be at the center of the display, or that a group of nodes be located close to each other, or that certain links must either go from top to bottom or left to right. Each of the above guidelines can be used to drive the algorithm design.

Aesthetics, however, often have significant impact on the interpretability of a tree or graph drawing, yet often result in conflicting guidelines. Some typical aesthetic rules include:

- minimize line crossings
- maintain a pleasing aspect ratio
- minimize the total area of the drawing

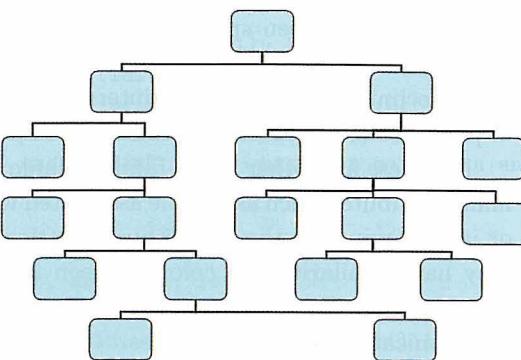


Figure 8.5. An example of visualizing hierarchies with a simple node-link diagram, using equal spacing per level.

- minimize the total length of the edges
- minimize the number of bends in the edges
- minimize the number of distinct angles or curvatures used
- strive for a symmetric structure

For trees, especially balanced ones, it is relatively easy to design algorithms that adhere to many, if not most, of these guidelines. For example, a simple tree drawing procedure is given below (sample output is shown in Figure 8.5):

1. Slice the drawing area into equal-height slabs, based on the depth of the tree.
2. For each level of the tree, determine how many nodes need to be drawn.
3. Divide each slice into equal-sized rectangles based on the number of nodes at that level.
4. Draw each node in the center of its corresponding rectangle.
5. Draw a link between the center-bottom of each node to the center-top of its child node(s).

Many enhancements can be made to this rather basic algorithm in order to improve space utilization and move child nodes closer to their parents.

Some of these include:

- a Rather than using even spacing and centering, divide each level based on the number of terminal nodes belonging to each subtree.
 - Spread terminal nodes evenly across the drawing area and center parent nodes above them.
 - Add some buffer space between adjacent nonsibling nodes to emphasize relationships.
 - If possible, reorder the subtrees of a node to achieve more symmetry and balance.
 - Position the root node in the center of the display and lay out child nodes radially, rather than vertically.

For large trees, a popular approach is to use the third dimension, supplemented with tools for rotation, translation, and zooming. Perhaps the most well-known of such techniques is called a *cone tree* [293]. In this layout, the children of a node are arranged radially at evenly spaced angles and then offset perpendicular to the plane. The two parameters critical to this process are the radius and offset distance; varying these influences the density of the display and the level of occlusion. Minimally they should be set so that

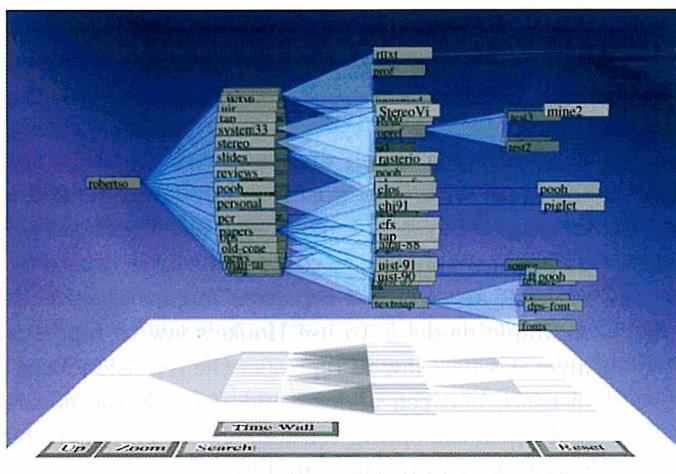


Figure 8.6. All example of a hierarchy displayed with a cone tree [293]. (Image © 1991 Association of Computing Machinery. Reprinted by permission, courtesy of PARC, Inc.)

separate branches of the tree do not fall into the same section of 3D space. One method to ensure this is to have the radius inversely proportional to the depth of a node in the tree. In this manner, nodes close to the root are significantly separated, and those near the bottom of the tree are closer together. An example is shown in Figure 8.6.

8.2 Displaying Arbitrary Graphs/Networks

Trees are just one type of a more general representation of relations called a graph. Technically speaking, a tree is a connected, unweighted, acyclic graph. Clearly, there are many other possibilities, including graphs with weighted edges, undirected graphs, graphs with cycles, disconnected graphs, and so on. Rather than give more algorithms specific to other classes of graphs, which could certainly fill more than a textbook, we will describe some general approaches for visualizing graphs in which the class or structure is not known, which we term an arbitrary graph. For our purposes, we will assume that the graph is undirected, though some of the techniques presented are easily extended to directed graphs. We will look at two distinct graph drawing approaches: node-link diagrams (building on the material from the previous section) and matrix displays. Readers interested in a broader or deeper exposure to graph drawing are directed to the vast amount of literature on this topic, some of which is listed at the end of the chapter.

8.2.1 Node-Link Graphs

Force-directed graph drawing methods use a spring analogy to represent the links, with node positions iteratively refined until the overall energy or stress of the system is minimized (see Figure 8.7). For each pair of connected nodes, there are two forces: f_{ij} , the force caused by the spring between them, and g_{ij} , an electrical repulsion force to keep nodes from getting too close. A simple model is to use Hooke's law to represent the spring force and an inverse square law to represent the repulsion force. If $d(i, j)$ is the Euclidean distance between nodes i and j , $s_{i,j}$ is the natural spring length (at rest), and k_{ij} is the spring tension, the x-component of the spring force between two nodes can be computed as

$$f_{ij}(x) = k_{ij} * (d(i, j) - s_{i,j}) * (x_i - x_j) / d(i, j).$$

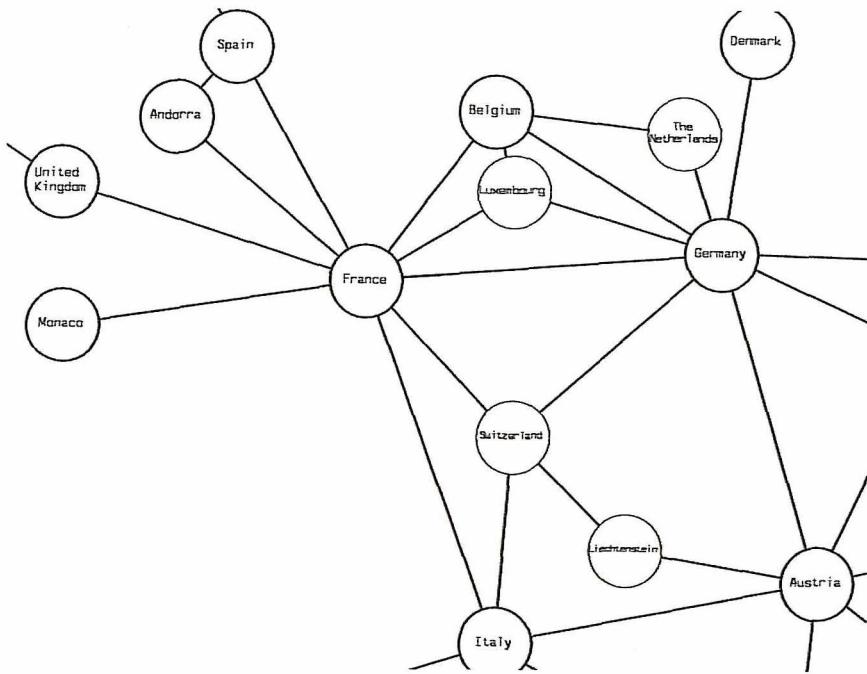


Figure 8.7. An example of a force-directed graph. The graph, showing relationships between countries of Europe, was generated with aiSee: <http://www.aisee.com>.

If r_{ij} is the strength of the repulsion between nodes i and j , the x -component of the repulsion force can be computed as

$$g_{ij}(x) = (r_{ij}/d(i, j)^2) * (x_i - x_j)/d(i, j).$$

Thus, one step of the position refinement process would calculate the sum of all the forces on each node (x -, y -, and z -components, as appropriate) and move its position proportional to that force. Clearly, once points have moved, all the forces need to be recalculated and another shift of positions made. To avoid oscillation, it is common to start with movements that are a significant percentage of the force and then use smaller and smaller step sizes to converge on the point where the forces are minimized. Initial positions can be assigned randomly. As it is quite possible to end up in a local, rather than a global, energy minimum, it is common to run the layout algorithm multiple times with different initial configurations to find the best of several computed configurations. The goodness of the layout can be computed based on the sum of the magnitude of forces on a given configuration.

Planar graph drawing techniques start with the assumption that the underlying graph is planar, e.g., it has no edge crossings. These algorithms have gotten a lot of attention, for several reasons. First, as the theory of planar graphs has a long history, there are many concepts that can be exploited from the literature. Second, as edge-crossings tend to make graphs difficult to read, it is a good strategy to minimize or eliminate such crossings. Finally, planar graphs tend to be sparse; Euler's formula indicates that a planar graph with n vertices has at most $3n - 6$ edges. Concentrating on planar graphs is not overly restrictive, as one can eliminate crossings by inserting dummy nodes at the crossings, perform the layout using a planar graph algorithm, and then remove the dummy nodes.

We will, in addition, assume that the graph is *connected*, e.g., there is a path from every node to every other node. Graphs that are not connected can be separated into subgraphs that can be drawn separately. A subgraph that is maximally connected (all nodes are connected) is a connected component of the graph. Other useful definitions include:

- A face is a partition of the plane isolated by a set of connected vertices.
- A neighbor set is a counter-clockwise listing of the vertices incident to a particular vertex.
- A planar embedding is a class of planar graph drawings with the same neighbor sets for each vertex. A planar graph can have an exponential number of such embeddings.
- A cutvertex is any node that causes the graph to be disconnected if it is removed.
- A biconnected graph is one without a cutvertex.
 - A block is a maximally biconnected subgraph of a graph.
 - A separating pair means two vertices whose removal causes a biconnected graph to become disconnected.
- A triconnected graph is one without a separating pair. A planar triconnected graph has a unique embedding.

We first need a strategy for determining if a graph is planar. Several such algorithms exist, though efficient ones have a very high degree of complexity and simple ones tend to be computationally expensive. We can start by simplifying the problem a bit. We do this by noting that a graph is planar

only if all of its connected components are also planar. Similarly, we can state that a connected graph is planar only if all its biconnected components are planar. Thus, we just need an algorithm that determines if a biconnected graph is planar or not.

The general reasoning of the algorithm is as follows. We will perform a divide-and-conquer approach by noting that if our graph contains a cycle such that no other cycle is present that doesn't contain an edge of the original cycle (e.g., there aren't cycles left when the edges involved in the original cycle are removed), what remain are paths that start and stop on one of the vertices of the cycle (called *attachments*). These *pieces* of the graph can be drawn either within the cycle or outside the cycle. Two such pieces *interlace* if they both start and end on nodes of the cycle, and the two ends of one piece are separated by one end of the other piece. To be drawn in a planar fashion, one of these interlaced pieces would need to be drawn inside the cycle, and the other on the outside. If we now create a graph of all the pieces, with an edge between two pieces if they interlace, as long as this graph is *bipartite* (separable into two sets of vertices such that no edge exists between members of the same set), the original graph is planar. Figure 8.8 shows examples of these components. Note that there are a couple of instances of interlacing among the parts.

If the graph contains more cycles after removing the edges of the original cycle, this means that one or more of the pieces contains a cycle (see the purple piece in Figure 8.8). In this case, we create a subgraph containing this piece and a section of the original cycle connecting the end points of the part, and recursively call the planarity test algorithm. The pseudocode

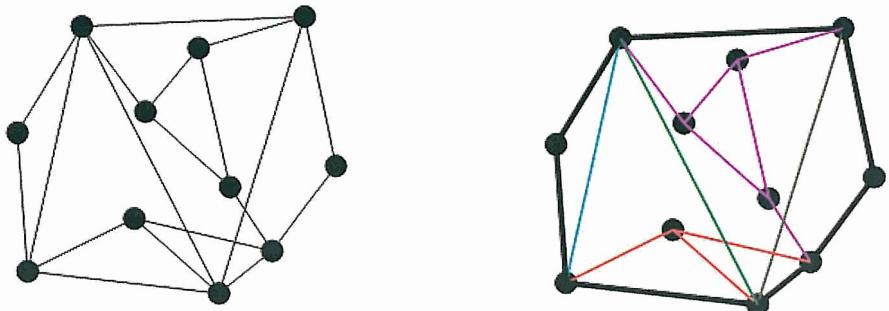


Figure 8.8.

An example of a biconnected graph, a cycle (in black), and the five pieces (in different colors).

for this algorithm is as follows [20]. Note that a *separating cycle* is one that generates at least two pieces.

Given a biconnected graph G and a separating cycle C:

1. Compute all the pieces of G with respect to C.
2. For each piece P that is not a simple path (e.g., that contains a cycle).
 - (a) Create graph G' consisting of P plus C.
 - (b) Create cycle C' consisting of a path through P plus the section of C joining the ends.
 - (c) Apply the algorithm to (G', C') . If the result is nonplanar, G is nonplanar.
3. Compute the interlacement graph I of the pieces of G.
4. If I is not bipartite, G is nonplanar; else G is planar.

If a graph is nonplanar, we can make it planar using the following strategy:

1. Determine the largest planar subgraph of the graph.
2. For the remaining vertices, place each within a face that minimizes the number of edge crossings.
3. For each edge crossing, break the edges into two parts each, and connect the broken ends to a new dummy vertex.

Once a graph has been either determined to be planar or has been augmented to achieve planarity, there are many possible strategies for generating a drawing. One such technique, called the *visibility approach* [20], consists of a two-step process. In the first step, called the visibility step, a *visibility representation* of the graph is formed. In such a representation, each vertex is depicted as a horizontal line segment, and each edge is depicted as a vertical line connecting the corresponding vertex segments. It should be clear that for a planar graph, it is always possible to draw such a representation without crossing edges other than where they meet the vertex segments. Obviously, many possible orderings of the vertex segments are possible; one strategy would be to arrange them to minimize the total length of the vertical connectors.

In the second step, called the replacement step, each vertex segment is collapsed to a single point, and each vertical connector is replaced by a

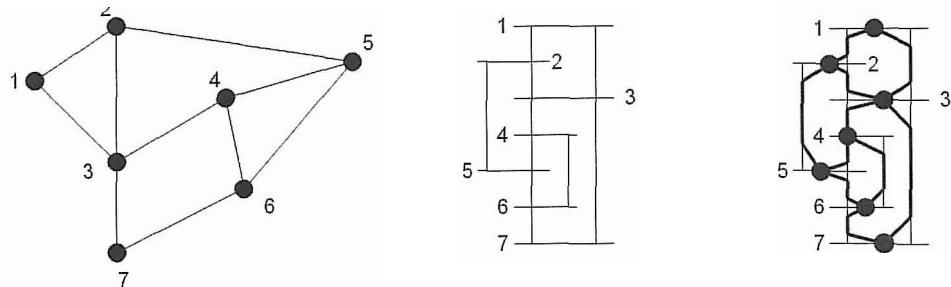


Figure 8.9. The stages of drawing a planar graph. From left to right: original graph, visibility representation, and replacement step.

polyline that follows the original edge as much as possible, with a segment at each end connecting the edge to its corresponding vertex. Many options exist for the replacement step, including the location of the nodes and the strategy used to form the connections (e.g., straight versus curved lines, single segment versus multiple segments). An example of the process is shown in Figure 8.9.

8.2.2 Matrix Representations for Graphs

An alternate visual representation of a graph is via an adjacency *matrix*, which is an N by N grid (where N is the number of nodes), where position (i,j) represents the existence (or not) of a link between nodes i and j . This may be a binary matrix, or the value might represent the strength or weight of the link between the two nodes. This method overcomes one of the biggest problems with node-link diagrams, namely that of crossing edges, though it doesn't scale well to graphs with large numbers (thousands) of nodes. Bertin [26] was one of the first researchers to investigate the power of this representation, using different reordering strategies to organize the rows and columns to reveal structures within the graph. The importance of the reordering is apparent in Figure 8.10, where each matrix represents the same eight-node graph. The two four-node cliques are clearly apparent in the second display.

There have been numerous algorithms proposed for reordering the rows and columns of the matrix to expose the most structure. Some are primarily user-driven, which would support ordering based on the values in one of the rows or columns as a starting point. Others are purely automatic, which

	a	b	c	d	e	f	g	h	
a		•	•			•			
b	•			•		•			
c	•				•		•	•	
d	•	•				•			
e			•				•	•	
f	•	•		•					
g			•		•			•	
h			•		•		•		

	p	q	r	s	t	u	v	w	
p		•	•	•					
q	•		•	•					
r	•	•				•			
s	•	•	•			•			
t				•		•	•	•	
u					•		•	•	
v					•	•		•	
w					•	•	•	•	

Figure 8.10. Two matrix displays of the same graph, using different orderings of nodes. Structure is more clearly present in the matrix on the right.

rely on some metric for evaluating a particular ordering and a strategy for generating orders to test. As in any optimization process, there is a good chance that finding the optimal ordering is NP-complete (namely, that no algorithm of polynomial or less complexity can be found). Thus, a number of heuristics have been proposed over the years that generally result in good orderings, especially for certain classes of graphs.

As an example, we can use a simplistic order evaluation strategy, namely to count the number of occurrences of matching elements in adjacent rows or columns. This tends to group nodes that link or don't link to a common node. In Figure 8.10, the left-most matrix has a score of 9 when counting only vertical neighbors, while the right-most matrix has a score of 20. By enumerating all possible orders, we can find the orderings that give the highest match score. For modest numbers of nodes, this would be an acceptable strategy, but since the number of possible orderings is on the order of $N!$, this approach does not scale well. Ordering of nodes is similar to the traveling salesman problem (TSP), where one tries to find a path that passes through a collection of cities without visiting any city more than once, while at the same time minimizing the total distance traveled. As this is basically the same problem as finding the ordering of the rows or columns of a matrix to minimize some metric, heuristic solutions that have been used for the TSP can also be employed here.

8.3 Other Issues

Once a basic visualization of a tree or graph has been developed, there are a number of additional considerations, primarily addressing the issue of interpretability. Two such important considerations will be elaborated upon in this section: labeling and interaction.

8.3.1 Labeling

Proper labeling of a visualization is crucial to allow a viewer to understand what is being shown. A map would be of little value without some form of labeling; similarly, a color-coded plot would be difficult to understand without some indication of the meaning associated with the colors. In tree and graph drawing, the problem of labeling is compounded, not only because of the potential for many nodes, but also because labels might also be needed for the links between nodes.

If there are only a small number of distinct labels, such as showing the type of link or a class associated with a node, it is best to use nontextual labels, such as the color, size, or shape of a node or the color, thickness, or line style of a link. This does not require much screen space and can usually be interpreted unambiguously even in the presence of modest amount of line crossing and node occlusion. However, if the number of distinct labels exceeds five or six, the likelihood of misinterpretation can become large. A key for interpreting the graphical attribute mapping is essential.

For small graphs, a common strategy for node labeling is to put the labels within the nodes, using rectangular or oval node shapes to accommodate the text. To avoid distorting the perception of the nodes, the size of the nodes should be dictated by the length of the longest label. For situations where the labels can be very long, one option is to use abbreviations or numeric labels, along with a key for interpretation. Viewers will eventually learn the correspondences between the shortened labels and their actual meaning. A similar strategy can be used for edge labeling, placing the labels near the center of the edge. For edges that are predominantly vertical, these should be to the left or right of the edge, while for predominantly horizontal edges, they should be above or below. Using a consistent strategy will reduce the potential for erroneously associating a label with the wrong edge.

At the other extreme, if there are a large number of distinct labels that need to be shown, or the labels themselves are quite long, it becomes readily apparent that simultaneous display of all labels will be ineffective. Several strategies have been developed to cope with this problem. A common solution is to only show labels in a small region of the graph, for example, within a certain radius of the cursor position. If the density of the display is too high, a distortion of the visualization may be required (see the next subsection) to provide more screen space for that section of the graph. An alternate to distortion that sometimes works is to rotate the graph to reduce the overlap between labels (see Figure 8.11). Another interesting solution proposed in [37] is to only show a random subset of the labels for a short

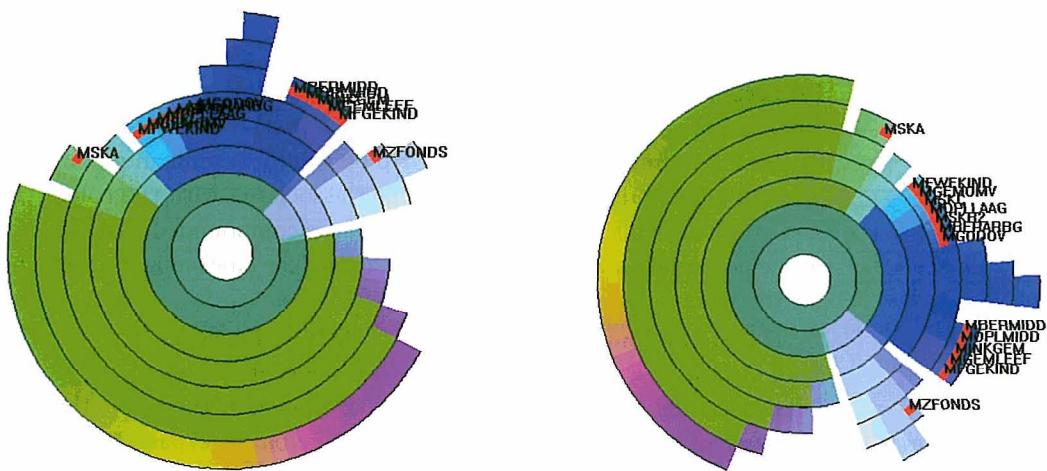


Figure 8.11. Improving the readability of labels via rotation. (Image from [412], © 2003 Palgrave Macmillan.)

period of time, and then switch to showing the labels for a different subset. The idea behind this approach is that the viewer's short-term memory will enable recall of a larger number of labels as compared to a static display, especially if this memory is refreshed on a regular basis.

8.3.2 Interactions

Even though Chapters 10 and 11 of this text are dedicated to interactions within visualization environments, there are a few interaction techniques that are most relevant to tree and graph visualization that will be described in this chapter. Some types of interaction, such as panning and zooming, are common to all types of visualization, and thus will only be briefly mentioned here for completeness. Others, such as focus+context, while applicable to a wide range of visualizations, have been primarily developed in the area of tree and graph visualization and will thus be described in more detail here.

Interactions with the virtual camera. Interactions such as panning, zooming, and rotation can be viewed as simple changes to the virtual camera being used to capture a segment of a scene. These allow the viewer to incrementally build up a mental model of the objects of the scene and their interrelationships. Operations of this type are often manually controlled, though

automated techniques such as data-driven fly-throughs and spinning of 3D objects can be automatically derived and presented.

Interactions with the graph elements. Most interactions of this type start with a *selection* operation, where one or more of the components of the graph are isolated for some action, such as highlighting, deleting, masking, moving, or obtaining details. For example, to declutter a graph one might select some nodes and drag them to a less-occupied section of the screen, while maintaining their links. Similarly, one might select and move or change the shape of a link to eliminate a crossing or improve the aesthetics of a graph. Selection may involve a single object, all objects within a specified region or distance, or a set of objects that satisfy a user-specified set of constraints (e.g., all nodes directly connected to a given node). One of the biggest problems with selecting elements in a graph occurs in dense regions of the drawing, where elements are so close together that unambiguous selection is difficult or impossible. This exposes the need for other types of interaction, such as zooming or the distortion techniques described later.

Interactions with the graph structure. There are two classes of interactions that are directed at the graph structure. The first class result in changes to the structure itself. For example, reordering the branches of a tree may expose relationships that were not apparent in the original ordering. Redrawing a graph with different weights on the constraints can generate graphs that make certain tasks easier to perform. Reordering the columns or rows in a matrix visualization can expose new features or relations within the data. Techniques within this class are often very specific to the type of graph being shown.

A second class of interactions associated with the graph structure comprises the so-called *focus+context* techniques, where a selected subset of the structure (*focus*) is presented in detail, while the rest of the structure is shown in low detail to help the viewer maintain context. These techniques are related to panning and zooming, without the loss of context. The most popular of these distortion techniques are the many variants on a fisheye lens, where the parts of the visualization falling within a focal region are enlarged using a nonlinear scaling, while the parts outside the focal region are proportionally shrunk to maintain their presence in the display. This distortion can be performed either in *screen space* (i.e., based on pixels) or in *structure space* (i.e., based on the components of the graph). It is the latter case that is more interesting in graph visualization, as we might, for example, enlarge one branch of a tree while reducing the size of other branches, or enlarge all links within three connections of a particular node in order to view

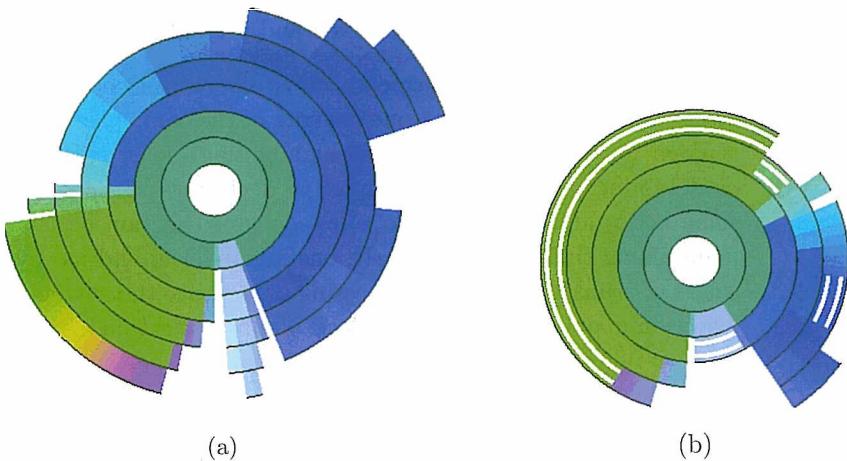


Figure 8.12. Some interaction operations on sunburst displays: (a) the blue subtree has been expanded, while the rest of the tree has been compressed; (b) several subtrees have been rolled up to simplify the display. (Image from [412], © 2003 Palgrave Macmillan.)

its neighborhood in more detail. An example of structure space distortion can be seen in Figure 8.12, where the blue subtree of Figure 8.11 has been angularly enlarged to enable easier exploration and interactive selection.

A technique that can be considered related to both of these classes is that of selective hiding or removal of sections of the graph. For example, once a branch of a tree has been thoroughly investigated, the user might want to remove it from the display to provide more space for the unexplored regions. In a sense, this can be seen as changing the structure (deleting a component), or as reducing the level of detail for the branch to its root. The terms *roll-up* and *drill-down* are often used to describe the process of hiding and exposing details in a visualization. Figure 8.12 shows several subtrees that have been rolled up, with the double white band informing the user that details exist under those nodes.

8.4 Related Readings

Robertson et al. [293] and Brian Johnson and Ben Schneiderman [176] introduce the concepts of cone trees and treemaps, respectively. John Stasko and Eugene Zhang [336] describe one of several variants on radial space-

filling techniques for tree visualization. The book, *Graph Drawing: Algorithms for the Visualization of Graphs* [20] is an excellent introduction to the field of graph drawing. *The Semeiology of Graphs* [26] by J. Bertin is the seminal work on reorderable matrix representations for graphs. Herman et al. [159] presents a survey of graph visualization and interactions with graphs. The paper by Leung and Apperley [232] contains a comprehensive survey of distortion techniques, many of which are applicable to tree and graph visualizations.

8.5 Exercises

1. Give some examples of how rules for graph drawing can conflict with each other.
2. Compare rectilinear and radial space-filling tree visualization techniques. Under what conditions, or for what tasks, is one better or worse than the other?
3. Compare node-link and matrix graph visualization techniques. Under what conditions, or for what tasks, is one better or worse than the other?
4. What is the smallest node-link graph (e.g., smallest number of nodes and links) that you can devise that is nonplanar?

8.6 Projects

1. Write a program that reads in a graph in the following format:

```
number-of-vertices number-of-edges
edge1_start edge1_end
edge2_start edge2_end
...
edgeN_start edgeN_end
```

Add a very simple drawing function that places the vertices in random positions and connects the vertices based on the edge list. Run the program several times with a data set of your design (it should have more than 10 nodes and 20 edges). What conclusions can you draw from observing the output?

2. Modify the above program to place the vertices at equal angles around a circle. Again, run the program several times and describe your observations. From these observations, can you propose a vertex-ordering algorithm that will generally result in less cluttered displays?
3. Write a program that will determine if a graph entered in the above format is connected, e.g., if there is a path from every node to every other node.
4. Write a program that will determine if a graph entered in the above format is biconnected, e.g., if removal of a single node will not disconnect the graph.
5. Assuming that the input graph represents a tree, and that all links are given in the order of (parent, child), write a program that will draw the tree as in Figure 8.5, where all nodes on the same level are evenly spaced. (Hint: in a single pass through the list of links, you should be able to assign each node to a level.)
6. Modify the above program to generate a *radial* layout, e.g., the layers are arranged as concentric circles with a radius proportional to the tree depth.
7. Modify either or both of the above programs to insert extra space between adjacent nonsibling nodes.
8. Write a program that generates the adjacency matrix A using the same data as in Project 1 or some other graph data. Use R-project (or your own code) to compute A^2 and draw it differentiating the values in the matrix using color (note that it may have values larger than 1). Explain what you see and the meaning of the numbers.