Unit-IV: Visualization of Different Structures

Relationship of Information

- While most of the visualization techniques 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.

Groups

1. **Venn Diagram:**
    1. **Description:** Overlapping circles represent relationships between groups, with each circle representing a distinct category.
    2. **Application:** Used to show the overlap or commonalities between different sets of entities.
2. **Clustered Bar Chart:**
    1. **Description:** Bars are grouped together based on categories, and each bar within a group represents a subcategory.
    2. **Application:** Effective for comparing values within different categories and subcategories.
3. **Pie Chart:**

    1. **Description:** A circular chart divided into slices, each representing a proportion of the whole.
    2. **Application:** Useful for displaying the distribution of a whole into different categories or groups.

4. **Heatmap:**
    1. **Description:** A matrix where colors represent values, and rows or columns are grouped based on categories.
    2. **Application:** Commonly used to visualize the intensity of relationships or values within a matrix.

5. **Dendrogram:**
    1. **Description:** A tree diagram that represents hierarchical relationships, with entities grouped based on similarity.
    2. **Application:** Commonly used in clustering analysis to show relationships within hierarchical structures.

6. **Bubble Chart:**
    1. **Description:** Circles represent groups, and the size of each circle indicates a certain value or dimension.
    2. **Application:** Useful for comparing multiple dimensions within different groups.

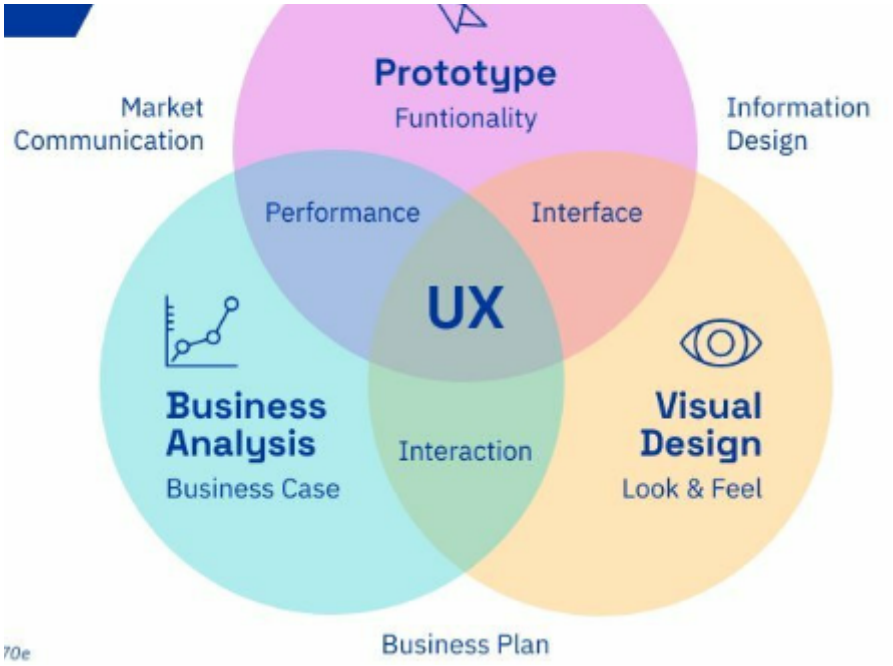
7. **Tree Map:**
    1. **Description:** Rectangles represent groups, with each rectangle divided into sub-rectangles based on a specific dimension.
    2. **Application:** Efficient for visualizing hierarchical data and the distribution of values within groups.

8. **Chord Diagram:**
    1. **Description:** Circles represent groups, and chords connect related groups, showing relationships between them.
    2. **Application:** Suitable for visualizing connections or interactions between multiple groups.

9. **Radar Chart:**
    1. **Description:** A chart with spokes radiating from the center, each representing a different category or group.
    2. **Application:** Useful for comparing values across multiple dimensions for different groups.

Venn Diagram

- A Venn diagram is a type of chart used to visualize the overlap between two or more datasets. It typically uses circles or ellipsis to illustrate relationships between segments, graphically highlighting how they are similar to or different from each other.
- Venn diagrams are widely used in many disciplines, most notably in mathematics, statistics, logic, teaching, linguistics, computer science and business.
- They are particularly effective at organizing and visualizing datasets, making the relationship easier to understand. Because of its powerful visual effect, the Venn diagram is a popular choice for business reports and presentations



Displaying Hierarchial 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.
- non–space-filling.

Space Filling Methods

- As the name implies, space-filling techniques make maximal use of the display space.
- This is accomplished by using justatpositioning 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.

Tree Maps- Rectangular Space Filling

- Treemaps 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.
- As mentioned, many variants on treemaps have been proposed and de veloped since they were introduced, including squarified treemaps (to reduce the occurrence of long, thin rectangles) and nested treemaps (to emphasize the hierarchical structure).

Algorithm

```
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.

Radial Space Filling

- Radial space-filling hierarchy visualizations, sometimes referred to as sunburst displays, 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 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.

Algorithm

```
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.

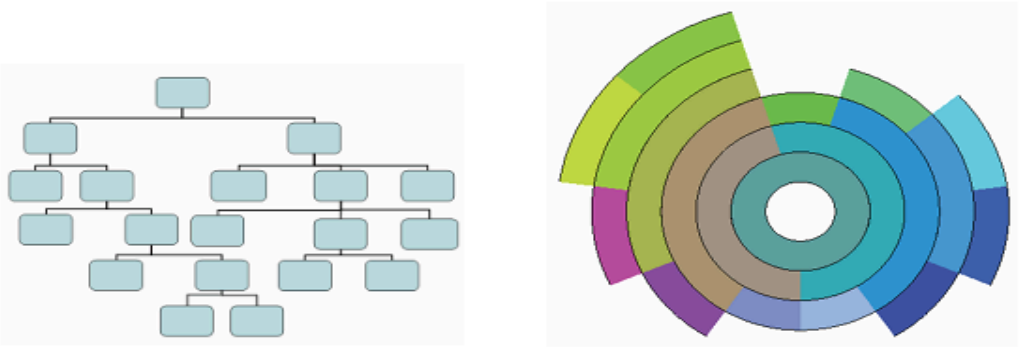


Figure 8.4. A sample hierarchy and the corresponding sunburst display.

Non Space Filling Methods

- The most common representation used to visualize tree or hierarchical re lationships 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

• 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

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).

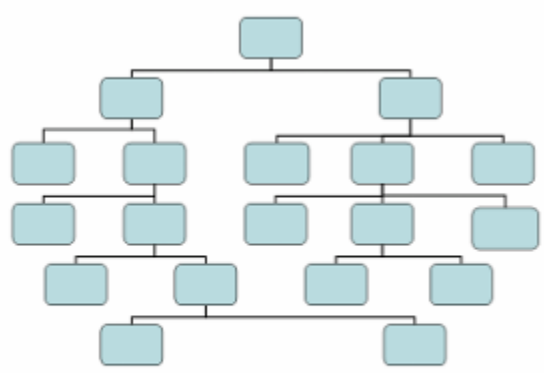


Figure 8.5. An example of visualizing hierarchies with a simple node-link diagram, using equal spacing per level.

- 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:

• 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 non sibling 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.

Cone Tree

- 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.

- 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 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.

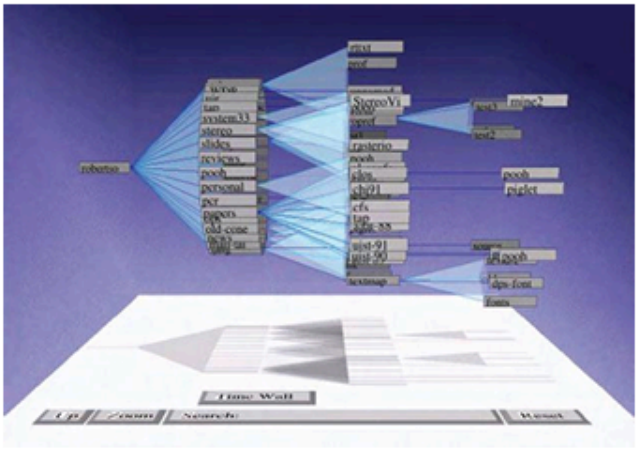


Figure 8.6. An example of a hierarchy displayed with a cone tree [293]. (Image © 1991 Association of Computing Machinery. Reprinted by permission, courtesy of PARC, Inc.)

Displaying Graphs and Networks

- 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.
- Types of graphs or networks include:

1. Node-link graphs
2. Planar Graphs

3. Matrix representation of graphs

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

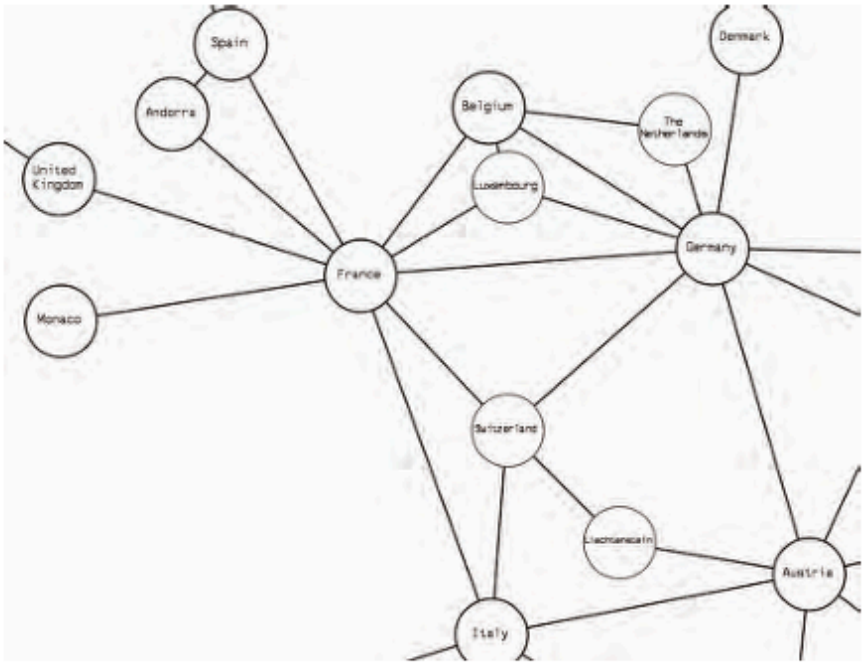


Figure 8.7. An example of a force-directed graph. The graph, showing relationships between countries of Europe, was generated with aiSee: http://www.aisee.com.

- 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

- $f_{ij}(x)=k_{ij} * (d(i,j) - s_{ij}) * (xi - xj)/d(i,j)$.
- 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 $g_{ij}(x)=(r_{ij}/d(i,j)2) * (xi - xj)/d(i,j)$
- 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.

Planar Graphs

- 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.

Definitions of Planar Graphs

- 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 cut vertex is any node that causes the graph to be disconnected if it is removed.
- A biconnected graph is one without a cut vertex.
- 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.

Algorithm to Represent a Planar Graph

- 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.

- 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.
- 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.
- 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.

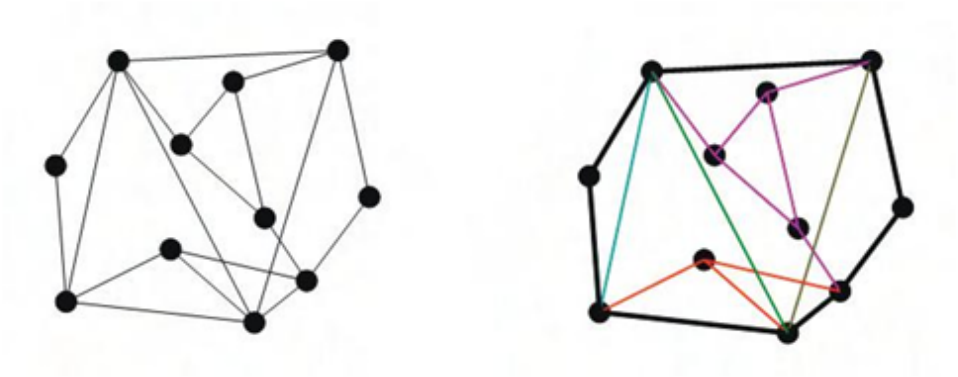


Figure 8.8. An example of a biconnected graph, a cycle (in black), and the five pieces (in different colors).

Pseudocode of Planar Graph Algorithm

- 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

Converting Non-Planar Graph to Planar Graph

- 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.

Drawing Planar Graphs

- Once a graph has been either determined to be planar or has been aug mented to achieve planarity, there are many possible strategies for generating a drawing.
- One such technique, called the visibility approach, 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.
- 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 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).

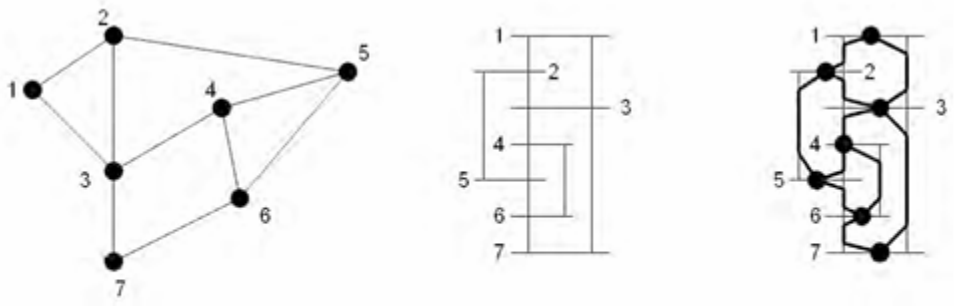


Figure 8.9. The stages of drawing a planar graph. From left to right: original graph, visibility representation, and replacement step.

Matrix Representation of 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.
- Bertin 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.

	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.

- 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 rely on some metric for evaluating a particular ordering and a strategy for generating orders to test.
- 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.

Issues with Visualizing Hierarchial Structures (Trees/Graphs)

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

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.

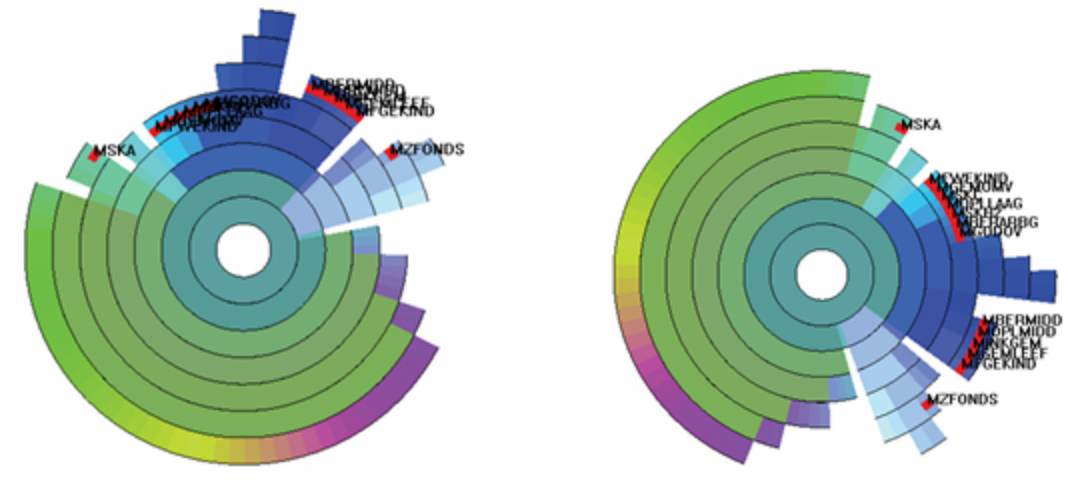


Figure 8.11. Improving the readability of labels via rotation. (Image from [412], © 2003 Palgrave Macmillan.)

Another interesting solution is to only show a random subset of the labels for a short 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.

Interactions

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 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 inter relationships. 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 Graph Elements

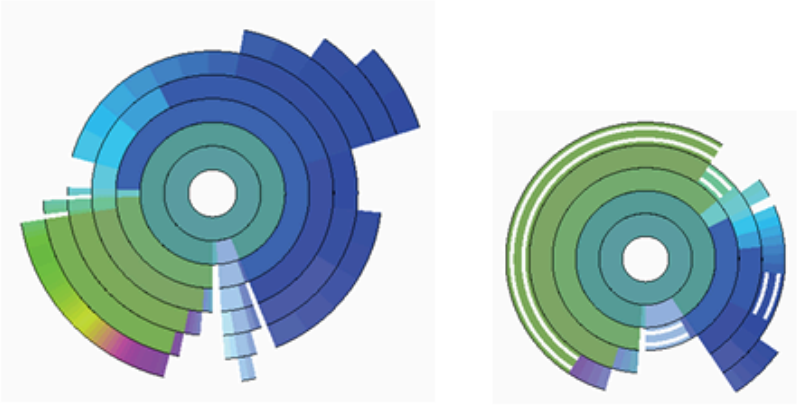
Most interactions of this type start with a selection operation, where one or more of the components of the graph are isolated for some action, such as highlighting, deleting, masking, moving, or obtaining details. For example, to declutter a graph one might select some nodes and drag them to a less-occupied 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 Graph Structures

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 com prises 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 dis tortion 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 en large all links within three connections of a particular node in order to view its neighborhood in more detail. An example of structure space distortion.

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.



(a) (b)

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.)

Cluster Visualization

1. Definition

Cluster visualization refers to the graphical representation of **grouped or clustered data**, where objects within a cluster share similar attributes, and different clusters exhibit distinct properties. The aim is to enable users to:

- Visually detect natural groupings in data.
- Identify relationships and separations between clusters.
- Explore patterns, correlations, and outliers.

2. Principles of Cluster Visualization

Cluster visualization operates on the following principles:

1. **Similarity Measurement:**
   Clusters are formed based on distance or similarity metrics (e.g., Euclidean distance, cosine similarity).
2. **Dimensional Reduction:**
   High-dimensional data is projected into 2D/3D space using methods such as PCA or MDS for visual inspection.
3. **Encoding Clusters Visually:**

- **Color:** Differentiates clusters.
- **Shape and Size:** Represent cluster centroids and density.
- **Transparency:** Conveys overlapping clusters or uncertainty.

3. Visualization Techniques for Clusters

a. Scatterplot Matrices (SPLOMs)

- Show pairwise relationships between attributes.
- Clustered regions appear as **densely populated patches** in subplots.
- Effective for detecting separations but limited in overlapping clusters.

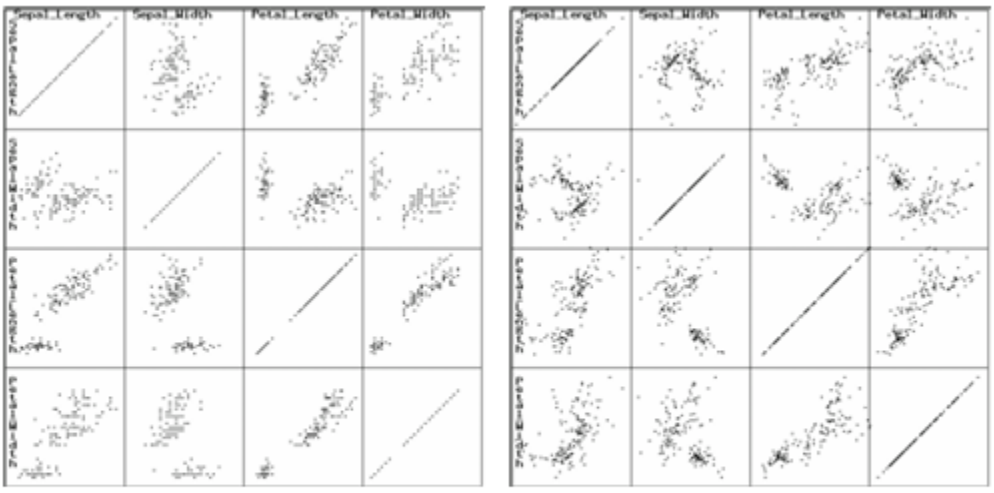


Figure 13.3. Iris data showing different degrees of clustering: (a) the original data; (b) has a 40% added noise factor [378]. (Image © 1997 IEEE.)

b. Parallel Coordinates

- Each variable is an axis; each record is a polyline crossing all axes.
- Clusters appear as **bundles of nearly parallel lines**.
- Enhancements like **hierarchical parallel coordinates** show summarized clusters via **opacity bands**.

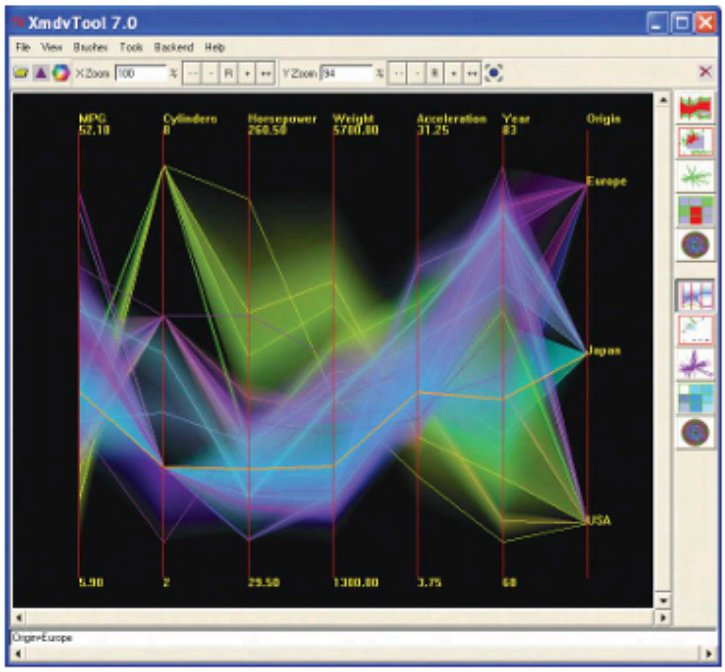


Figure 14.4. An example of hierarchical parallel coordinates in XmdvTool. Each line represents the center of a cluster, and the opacity bands around each centerline indicate the extents of the cluster in each dimension. The opacity near the center conveys the cluster population. Users can drill-down or roll-up in the data hierarchy, using a structure-based brush (see Figure 14.5).

c. Glyph-Based Techniques

- Represent clusters using composite icons (e.g., Chernoff faces, star plots).
- Each glyph encodes multiple variables for cluster comparison.

d. Vectorized RadViz (VRV)

- Converts multivariate dimensions into binary sub-dimensions.
- Clusters are visualized as separated regions in a circular layout.

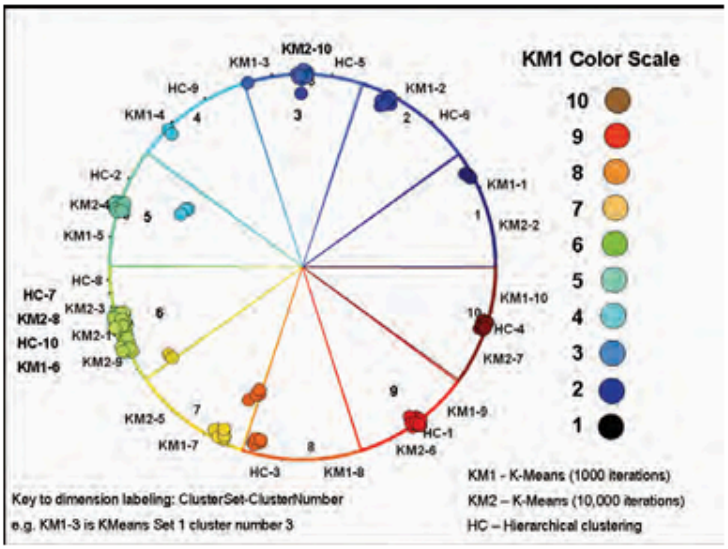
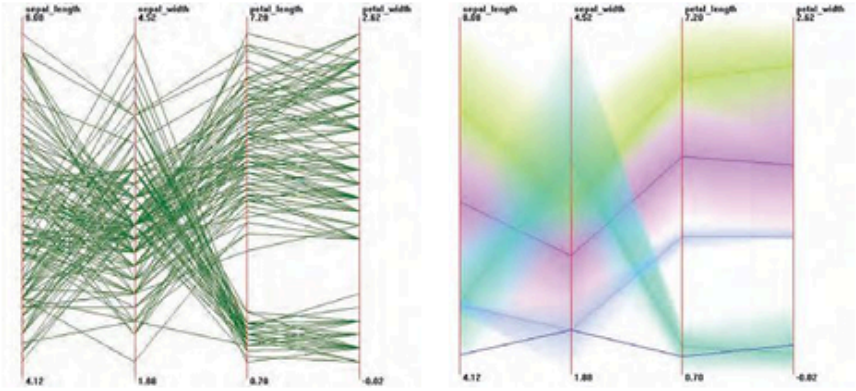


Figure 7.5. Vectorized RadViz, formed by splitting each dimension into multiple dimensions to create a binary representation for each data record. In this case, each cluster set is separated into multiple dimensions, where each dimension represents a cluster in each cluster set [315].

e. Aggregated Parallel Coordinates

- Used for large data sets where individual records are replaced with **cluster centers** and **extents**.
- Improves readability while preserving distribution information.



(a) (b)

Figure 2.5. The Iris data set in parallel coordinates: (a) the original data; (b) the centers and extents of clusters after aggregation. Each axis in parallel coordinates represents a dimension, with each record being drawn as a polyline through each of the coordinate values on the axes.

4. Applications of Cluster Visualization

Application Area	Purpose / Use	Visualization Techniques
Bioinformatics	To group genes or proteins by expression similarity	Scatterplot matrices, heatmaps, RadViz
Market Segmentation	To identify customer groups with similar behavior	Parallel coordinates, cluster maps
Document Analysis	To visualize topics or themes in large text corpora	Self-Organizing Maps (SOMs), Themecapes
Network Analysis	To detect communities within graphs	Force-directed layouts with cluster coloring
Visual Analytics	To summarize high-dimensional spaces for exploration	XmdvTool, hierarchical cluster visualization
Outlier Detection	To distinguish anomalies from normal cluster groups	Scatterplots, glyphs, and radial displays

5. Advantages of Cluster Visualization

- Enables **intuitive detection** of structure and anomalies.
- Facilitates **multi-dimensional exploration**.
- Supports **interactive analysis** via brushing, linking, and hierarchical roll-ups.
- Enhances **pattern recognition** in large or noisy data sets.

Metaphorical Visualization

1. Definition

Metaphorical visualization uses **real-world analogies or metaphors** to represent abstract data visually.
Instead of directly mapping data values to geometric primitives (like bars, lines, or points), the visualization borrows familiar **physical or conceptual metaphors** — such as landscapes, planets, buildings, or trees — to help users intuitively interpret complex information.

2. Purpose and Motivation

- To **enhance comprehension** of abstract or multidimensional data.
- To leverage **human familiarity** with physical environments.
- To provide **intuitive exploration** in visual analytics.
- To **engage users** through storytelling and memorable design.

Humans are naturally skilled at interpreting spatial, hierarchical, and physical relationships — metaphorical visualization builds upon these capabilities.

3. Characteristics of Metaphorical Visualization

Feature	Description
Analogical Mapping	Data attributes are mapped to features of a familiar metaphor (e.g., buildings → entities; height → magnitude).
Cognitive Accessibility	Reduces learning curve by using intuitive metaphors.
Engagement	Encourages exploration through immersive visuals.
Scalability	Often limited — works best for moderate data sizes.
Perceptual Coherence	Relies on consistent, meaningful mapping (e.g., larger = more important).

4. Types of Metaphorical Visualization

a. Landscape Metaphors

- Represent data as terrains or topographic surfaces.
- Peaks = high values, valleys = low values.
- Useful in representing data distributions and similarity clusters.
  Example: *ThemeScapes* visualize document clusters as "mountain ranges" based on topic density.

b. City Metaphors

- Represent information spaces as **urban layouts**:
  - Buildings = entities (files, classes, webpages).
  - Building height = importance, size, or frequency.
  - Streets = relationships or hierarchies.
- Commonly used in **software visualization** and **website analytics**.

Example: *CodeCity* visualization of software systems — classes as buildings, packages as districts.

c. Tree and Botanical Metaphors

- Use natural growth patterns to represent hierarchies or evolutions.
- Branches = subcategories; leaves = data elements.
- Example: "TreeMap" or "Information Garden" metaphors.

d. Solar System and Planetary Metaphors

- Used for **relationship-centered visualization**.
    - Central node (sun) = key entity.
    - Orbiting nodes (planets) = related items.
    - Distance = degree of relationship strength.
- Common in web analytics, bibliometric visualization, and recommendation systems.

Example: "Planetary View of Document Similarity" — Core topic surrounded by related subtopics.

e. Metaphors in Virtual Reality (VR)

- Immersive metaphors such as **"data rooms"**, **"knowledge galaxies"**, and **"data oceans."**
- Allow navigation through high-dimensional data via natural spatial movements.
- Used in visual analytics and large-scale simulations.

5. Advantages

- Simplifies understanding of **complex data spaces**.
- Engages users emotionally and cognitively.
- Facilitates memory retention and pattern recall.
- Promotes exploratory analysis through familiar spatial organization.

6. Applications

Domain	Use Case	Visualization Type
Software Engineering	Visualizing program structure	City and building metaphors
Document Analysis	Topic similarity mapping	Landscape metaphor (ThemeScapes)
Web Analytics	Website hierarchy	Solar system metaphor
Biological Data	Genetic relationships	Tree and branch metaphors
Financial Data	Market dynamics	Terrain or skyline metaphor
Education and Storytelling	Conceptual mapping	Virtual or 3D metaphorical spaces