PV251 Visualization

Autumn 2024

Study material

Lecture 8: Trees, graphs, networks, text, documents

Trees, graphs, networks

Until now, we have dealt with visualization techniques from the point of view of displaying data values and their attributes. Now we will look at another important application area of visualization - providing information about the relationships between data, i.e., how data items or records relate to each other.

Relationships between data can take several forms:

- Hierarchical relations (section vs. subsection, parent/child, …)
- Connectivity, interconnection (e.g., how cities are connected by roads, how computers interconnected into a network, …)
- Derivation (for example, a sequence of steps or phases)
- Shared classification
- Similarity between values
- Similarity between attributes (e.g., spatial, temporal, …)

Relationships can be simple or complex: one-directional or bi-directional, unweighted or weighted, etc. Additionally, relationships between data can provide much more information than just what is contained directly in the records.

Applications that display relational information can be very different - from categorizing animal species, through examining a document archive, to monitoring a terrorist network.

In this lecture we will describe a set of techniques for visualizing relational information. However, our overview will be only a fraction of what exists in this area, as a number of extensive publications are devoted to the visualization of trees and graphs.

Visualization of hierarchical structures

Hierarchical structures (often referred to simply as "trees") are some of the most common structures used to store information about the relationships between data. Many visualization techniques were therefore created to display these structures and preserved relationships. These techniques and their algorithms can be divided into two classes:

- **Space-filling** – make maximum use of screen space

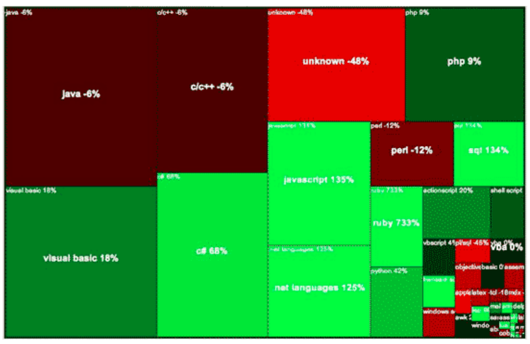
- **Non-space-filling** – maximum use of space of the output device is not their main criterion

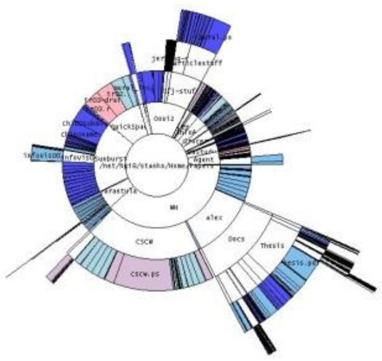
When studying the techniques for displaying hierarchical structures, we will focus on these two types of algorithms.

Space-filling methods

As the name suggests, these techniques try to make the most use of available screen space. This is often accompanied using juxtapositioning to display relations. The two most common approaches to generating this type of hierarchy are **rectangular** and **radial layouts**.

Among the most popular forms of space-filling layout belong treemaps and their various variants. These are an example of a rectangular space layout.





Representatives of the second class, i.e., the radial layout, are, for example, the so-called "sunburst displays" (in the shape of the sun). Here, the root of the hierarchy is displayed in the center, and nested ring layers are used to display additional hierarchy layers.

Treemaps

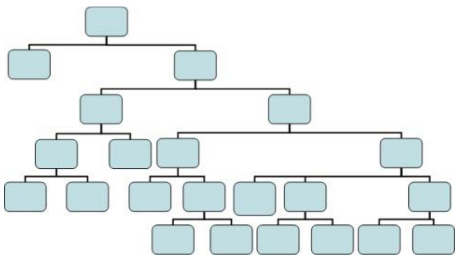
In their basic variant, the rectangle is recursively divided into segments, alternating the horizontal and vertical division of the rectangle. The division is derived from the population of subtrees at a given level. As an example, consider the following pseudocode:

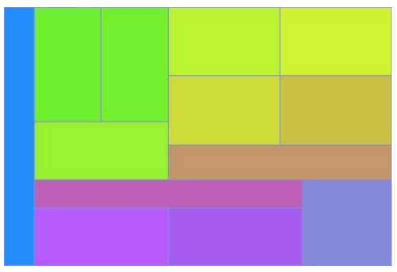
Notation:

```
Width = width of the rectangle
Height = height of the rectangle
Node = root node of the tree
Origin = position of the rectangle (e.g., [0, 0])
Orientation = direction of division - alternating horizontal and vertical
division
```

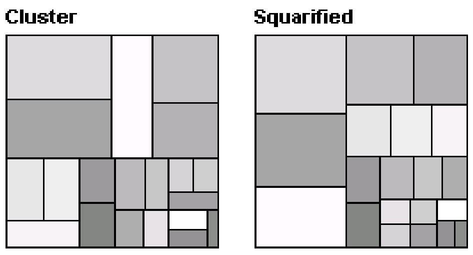
```
treemap (Node n, Orientation o, Position orig, Width w, Height h)

treemap (Node n, Orientation o, Position orig, Width w, Height h)
   if n is leaf node (has no children)
      draw_rectangle(orig, w, h);
      return;
   for each child of node n (child_i) get number of leaf nodes in subtree;
   sum the total number of leaf nodes of n;
   calculate the percentage of leaf nodes in each subtree (percent_i);
   if orientation is horizontal
      for each subtree
            calculate origin offset based on origin and width (offset_i);
            treemap(child_i, vertical, orig + offset_i, w * percent_i, h);
   else
      for each subtree
            calculate origin offset based on origin and height (offset_i);
            treemap(child_i, horizontal, orig + offset_i, w, h * percent_i);
```

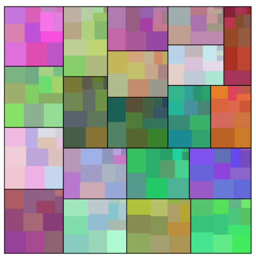
The following figure shows an example of a visualization of a given hierarchy in the form of a treemap created using the algorithm described by this pseudocode.





Many variants of treemaps have been developed, such as squarified treemaps (to reduce long, thin rectangles) or nested treemaps (to enhance the hierarchical structure).





Radial layout

Methods for displaying hierarchy in radial layout, often referred to as "sunburst displays", have the root of the hierarchy located in the center of the radial display and the individual layers of the hierarchy are represented by concentric rings. Each ring is divided based on the number of nodes in a given level of the hierarchy. These techniques use a similar strategy as treemaps – the number of leaf nodes in a given subtree determines the size of the screen that allocated for that subtree.

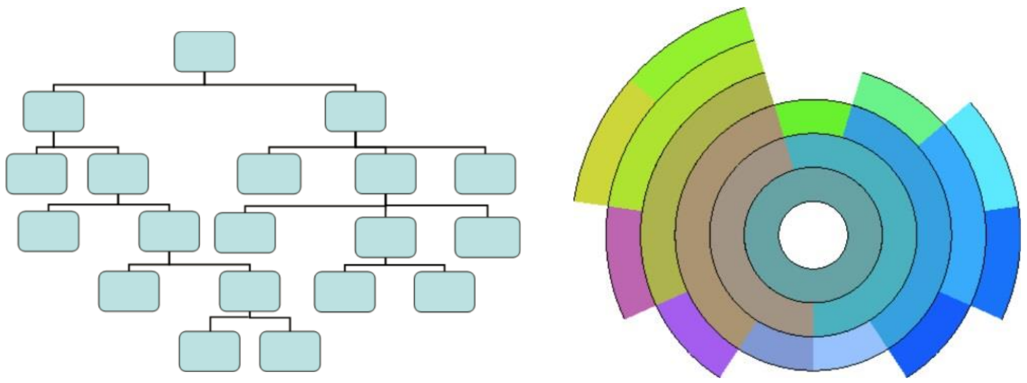
However, while treemaps dedicate most of the screen space to displaying leaf nodes, this radial layout technique also displays inner nodes. The pseudocode of this algorithm is as follows:

Notation:

```
Start = starting node angle (initially 0)
End = end angle of the node (initially 360)
Origin = position of the center of the radial display (e.g., [0, 0])
Level = current level of the hierarchy (initially 0)
Width = thickness of each radial strip - based on the maximum hierarchy
depth and the size of the display
```

`sunburst (Node n, Start st, End en, Level l)`

```
sunburst (Node n, Start st, End en, Level l)
    if n is leaf node (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 leaf nodes in subtree;
    sum the total number of leaf nodes of n;
    calculate the percentage of leaf nodes in each subtree (percent_i);
    for each subtree
      calculate the start / end angle based on the size of the subtrees,
      their arrangement, and angle ranges;
      sunburst (child_i, st_i, en_i, l + 1);
```



Using color in hierarchical techniques

For the aforementioned techniques, as well as for other space-filling techniques, color can be used to highlight many attributes, such as the values of each node (e.g., for classification) or it can enhance the display of hierarchical relationships (e.g., siblings and their predecessors can have a similar color/hue, which is observable also in the previous picture).

Other data properties can be communicated, for example, by using symbols and other markings placed in rectangular or circular segments.

Non-space-filling methods

One of the most common representations of hierarchical relationships are node-link diagrams. The structure of the organization, family trees, or pairings in tournament matches are just a few examples of common applications of these diagrams. The appearance of these trees is most affected by two factors: the **degree of branching** (the maximum number of siblings per parent) and the **depth** (determined by the farthest node from the root).

Different tree types are limited in one or both factors, such as binary trees or trees that allow only three or four layers.

When designing algorithms for plotting node-link diagrams (not just trees), we must consider three categories of rules, which are often contradictory:

- Rendering conventions
- Restrictions
- Aesthetics

Conventions can include edges, which can be defined by only one straight line, a set of lines, polylines, or general curves. Another convention may be to place the nodes in a fixed grid or to place the siblings in the same vertical position.

Restrictions may include a request to place the node in the center of the screen, a request to place a group of nodes in close proximity to each other, or a request to orient the lines in certain direction.

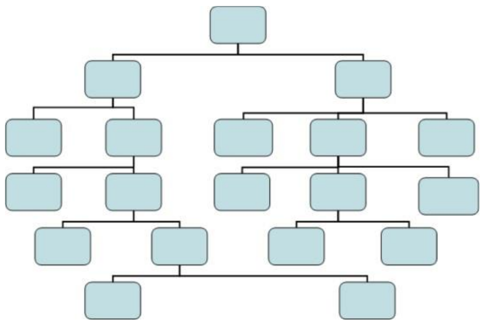
Aesthetics has a significant influence on the interpretation of a given tree or graph in general. Some typical aesthetic rules include:

- Minimize line crossing
- Maintain a "reasonable" aspect ratio
- Minimize the total area used for rendering
- Minimize edge length
- Minimization of the number of edge bends
- Minimize the number of different edge angles or curves
- Try to maintain symmetry

For trees, especially the balanced ones, it is relatively easy to design algorithms that comply with most of these rules. An example of this is the rendering procedure, which we will now show:

1. Divide the drawing area into horizontal layer of the same height – this division is derived from the depth of the tree
2. For each level of the tree, determine how many nodes must be drawn

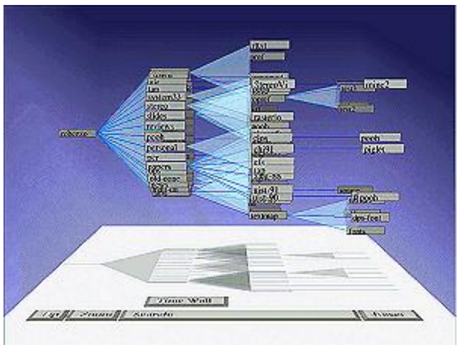
3. Divide each layer into rectangles of the same size – derived from the number of nodes at a given level
4. Draw each node in the center of the corresponding rectangle
5. Draw a connecting line from the center of the bottom edge of each node to the center of the top edge of its descendant(s)



This basic algorithm can be enhanced by several optimizations that improve the use of screen space by moving children closer to their parents. These optimizations include, for example:

- Instead of evenly dividing the space and centering, each level is divided according to the number of end nodes belonging to the given subtree.
- Even distribution of end nodes in the space intended for drawing and centering of their parent nodes.
- Adding additional separating spaces between adjacent nodes that are not siblings. The goal is to highlight relationships.
- If possible, reorganize the subtrees of nodes to achieve greater symmetry and balance.
- Place the root node in the center of the screen and radially place the children around it.

For large trees, a popular approach is to use a third dimension, complete with tools for rotation, translation, and zooming. Probably the best known of such techniques are the so-called **cone trees**. In this layout, the descendants of the node are evenly distributed radially around the parent and then shifted perpendicularly into a plane.



The two main parameters are the radius and the offset distance – their change affects the display density and the degree of overlap.

At a minimum, these trees should meet the condition that their separate branches are not located in the same place in 3D space. One method of achieving this condition is to determine the radius using the inverse ratio to the depth of the node in the tree. This way, the nodes near the root are closer to each other.

Displaying arbitrary graphs/networks

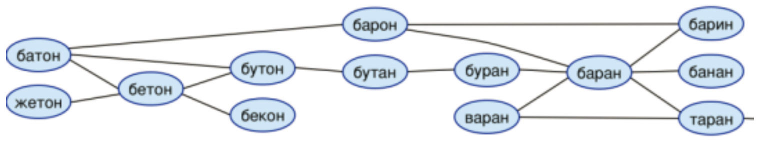
Trees are just one type of more general representation of relationships between data, called **graph**. The tree is an example of a joint unweighted acyclic graph. Obviously, there are many different graph variations, such as weighted edge graphs, non-oriented graphs, graphs containing cycles, disconnected graphs, etc.

Rather than describing the algorithms for these individual types of graphs, we will deal with the visualization of graphs whose structure is not known in advance. We call them **arbitrary graphs**. In our case, we will assume that the graph is unoriented, although most of the presented techniques can be easily extended to oriented graphs. We will focus on two different approaches to rendering:

- **Node-link diagrams**
- **Matrix displays**

Node-link graphs

The force-directed graphs described in the last lecture use the analogy of springs to represent links, where the positions of nodes are iteratively adjusted until the value of so-called *stress* is minimized (e.g., multidimensional scaling). An example is shown in the figure.



For each pair of connected nodes, two forces are calculated: f_{ij} - the force derived from the string between them, g_{ij} - electrical resistance preventing the nodes from getting too close to each other. A simple model is to use Hooke's law to represent the force f_{ij} and the inverse square law to represent g_{ij}. If the Euclidean distance between nodes i and j is denoted as d(i,j), s_{ij} is the length of the string (at rest) and k_{ij} is the tension of the string, then the x-component of the force f_{ij} between these nodes is calculated as

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

If r_{ij} denotes the force of resistance between nodes i and j, then the x-component of the force g_{ij} is calculated as:

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

Each step of the position adjustment process calculates the sum of all forces for each node (x, y, and z-components) and shifts the position accordingly in proportion to the resulting force. Of course, after adjusting the position, we must recalculate all the forces and the position shifts again. It is necessary to avoid oscillations, so we adjust the force to converge to the point where the forces are minimized. Starting positions are assigned randomly. It is likely that the point will end up in a local rather than a global energy minimum. Therefore, it is common practice to run the algorithm several times in a row with different initial configurations and then select the best final configuration. The "suitability" of the location of points in a given layer can be calculated based on the sum of the magnitudes of the forces in a given configuration.

Planar (planar) graphs

Techniques for plotting 2D graphs are based on the assumption that the basic graph is **planar**, i.e., its edges do not intersect. Algorithms for planar graphs are very popular for many reasons. Firstly, the planar graph theory has a long history, and there are several studies on the subject. Secondly, edge crossings generally complicate the interpretation of graphs, so it is advantageous to minimize these crossings, preferably to eliminate them completely. Finally, planar graphs are often sparse: Euler's theorem for planar graphs says that at n vertices they can have at most 3n-6 edges.

Limiting oneself to working with planar graphs is not as restrictive as it would seem. This is because graphs with crossing edges can be converted to a planar graph as follows. We place dummy nodes at the intersections of the edges, run the algorithm for planar node positioning, and then remove the false nodes again.

In addition, we will assume that the graph is **connected**. That means there is a path from each node to all other nodes. Graphs that are not connected can be divided into subgraphs that are plotted separately as connected.

A **maximal connected** subgraph (all nodes are interconnected) is called a **connected graph component**.

Other definitions:

- **Face** – the part of the plane enclosed by a set of connected vertices.
- **Neighbor set** – a set of vertices that are adjacent to a given node (in counter clockwise order).
- **Planar embedding** – a class of planar graph representations with the same neighbor set for each vertex. A planar graph can have exponentially many such embeddings.
- **Cutvertex** – any node which, when removed, makes the graph disconnected.
- **Biconnected graph** – graph without cutvertices.
- **Block** - maximum biconnected subgraph of the graph.

- **Separating pair** – a pair of vertices, the removal of which will cause the biconnected graph to become disconnected.
- **Triconnected graph** – graph without separating pairs. The planar triconnected graph has a unique embedding.

We must first determine the strategy according to which we decide whether the given graph is planar. There are several algorithms that solve this problem, but the time-efficient ones are very complex and, conversely, simple algorithms are computationally intensive.

Therefore, we will simplify the problem a bit. We say that a graph is planar if all its connected components are also planar. Similarly, we can say that a connected graph is planar only if all biconnected components are planar. Therefore, we only need an algorithm to determine whether the biconnected graph is planar or not.

The algorithm works as follows. We will use the divide and conquer approach. If our graph contains a cycle that has no other cycle that does not contain the edge of the original cycle (in other words, if we remove the edges of the original cycle, no more cycles remain), then there are paths (called *pieces*) that begin and end in one of the vertices of this cycle (called *attachments*). These arise as follows: Two edges e and e' belong to the same piece if there is a path starting at e and ending at e' that does not contain the vertices of the cycle as its "inner" vertices.

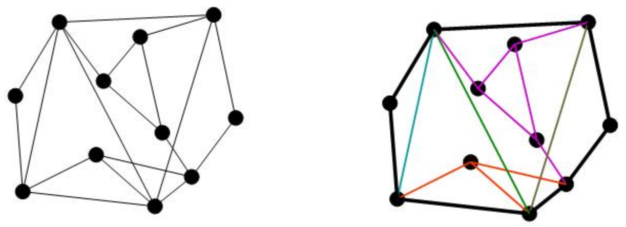
The piece can be:

- One edge between two cycle vertices
- A continuous graph that contains at least one vertex that does not lie in the cycle

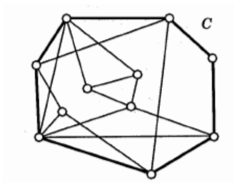
Two such pieces of a graph are "interlaced" if they both begin and end at the nodes of the cycle, and the two ends of one piece are separated by the end of the other piece. In order to draw these pieces planarly, one of these pieces must be drawn inside the cycle and the other outside.

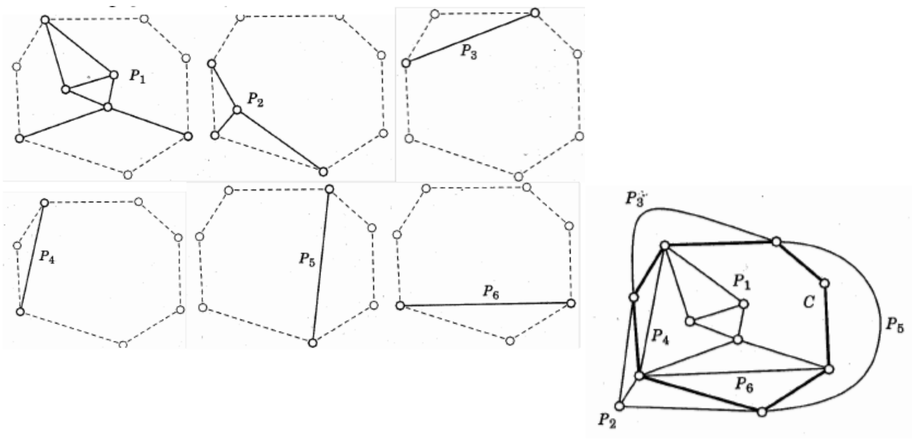
If we now create a graph of all the pieces, where two intersecting pieces are separated by an edge and such a graph is bipartite (divisible into two sets of vertices in such a way that there is no edge between members of the set), then the original graph is planar.

The figure shows an example of a biconnected graph. On the right, the cycle is shown in black, and five pieces are shown in color.



If the graph contains multiple cycles after removing the edges of the original cycle, it means that one or more pieces contain a cycle (see the pink piece in the figure). In this case, we create a subgraph containing this piece and the part of the original cycle connecting the endpoints, and recursively call an algorithm to test the planarity of the graph.





For completeness, we will now introduce the pseudocode of this algorithm. Let's call a **separating cycle** a cycle that generates at least two pieces.

```
Let us consider a biconnected graph G and a separating cycle C.
   1. Calculate all pieces of G with respect to C.
   2. For each piece P that is not a path (contains a cycle)
         a. We create a graph G’ consisting of P and C.
         b. We create a cycle C’ consisting of a path through P and a part
            of C connecting the ends.
         c. We apply this algorithm to (G’, C’). If the result is non-
            planar, then G is also non-planar.
   3. Compute the conflict graph I of pieces of G.
   4. If I is not bipartite, G is non-planar; otherwise, G is planar.
```

If the graph is non-planar, we can make it planar using the following strategy:

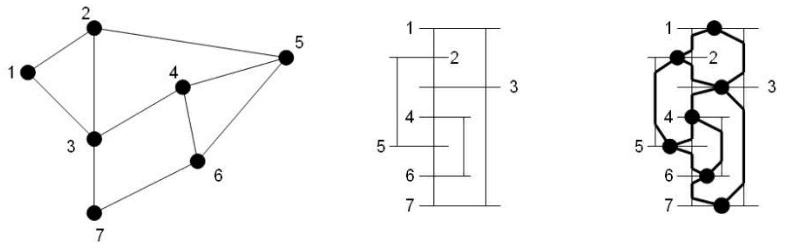
1. Determine the largest planar subgraph of the original graph.
2. Place the remaining vertices in the faces in a way that minimizes the edge crossings.
3. For each intersection of the edges, divide the corresponding edges into two parts and create a new "dummy" vertex at the intersection.

Rendering planar graphs

After marking a given graph as planar or expanding it to achieve planarity, we can use many techniques to generate a representation of these graphs. One of these techniques is the so-called **visibility approach**, which consists of two steps:

1. **Visibility step** – we will create the so-called visibility representation. In this representation, each vertex is drawn as a horizontal line segment, and each edge is drawn as a vertical line connecting the corresponding vertex segments. Obviously, for planar graphs, it is always possible to draw such a representation without intersecting edges except for those in which the vertex segments meet. Obviously, there are many possible arrangements of line segments – one strategy may be to arrange the segments to minimize the length of the vertical joints.

2. **Replacement step** – each segment corresponding to the vertex is "shrunk" to a single point and each vertical line is replaced by a polyline, which tries to preserve the "appearance" of the original edge as much as possible. Again, there are several variations for this step, such as using curved lines, using one vs. more segments, etc.



Matrix representation of graphs

An alternative visual representation of the graph is the so-called adjacency matrix, which is a grid of size NxN, where N is the number of nodes. Position (i, j) represents the presence or absence of a link between nodes i and j. The adjacency matrix may be a binary matrix or the value at a given position may correspond to the strength or weight of the link between the corresponding two nodes.

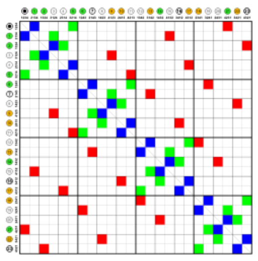
This method overcomes one of the biggest problems of node-link diagrams, which is the crossing of edges. However, it is not suitable for very large graphs (with thousands of nodes).

Bertin was one of the first scientists to study the advantages of this representation. In particular, he focused on various strategies for organizing rows and columns to reveal interesting structures in the chart. The impact of the rearrangement is clear from the figures, where each of the matrices represents the same graph with eight nodes.

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

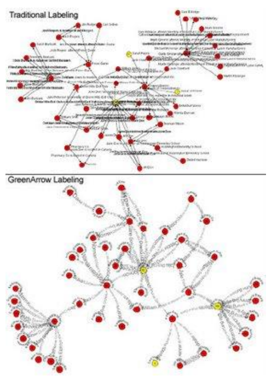
	p	q	r	s	t	u	v	w
p		●	●	●				
q	●		●	●				
r	●	●		●				
s	●	●	●		●			
t				●		●	●	●
u					●		●	●
v					●	●		●
w					●	●	●	

There are many different algorithms for reorganizing rows and columns. Some of them are primarily user-controlled, others are fully automatic. As with any optimization problem, finding the optimal order of rows and columns is an NP-complete problem (more precisely, no polynomial or faster algorithm was found). Therefore, it was necessary to implement several heuristics that provide quality results, especially for some classes of graphs.



Labeling

Appropriate labeling of the visualization is necessary to understand what data we are actually displaying. For example, a map without labels would not be very helpful. Similarly, a graph using the coloring of individual elements representing the data would not be understandable if we did not define what the colors represent. When plotting trees and graphs, plotting labels is not easy, especially since the tress can contain many nodes and individual edges can also be marked.



If we need only a small number of different labels, it is better to use "non-text" labels, such as color, size, or shape of the node, or the color, thickness, or style of the line representing the edge. This does not take up the display area, and usually these representations are interpreted unambiguously even in the case of many crossing edges or overlapping nodes.

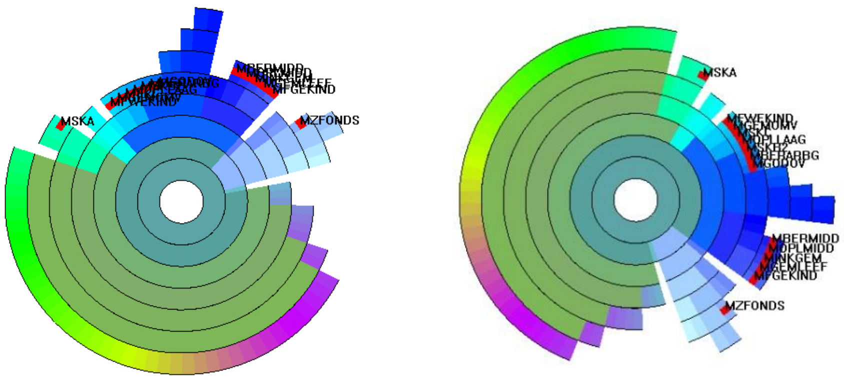
However, when the number of different labels exceeds five or six, the probability of misinterpretation is relatively high.

For small graphs, a common strategy for placing node labels is to insert them directly into the nodes - rectangular or oval node shapes are used to contain the text. To avoid misperception of individual nodes, the size of the nodes should be set according to the length of the longest label. In cases where the labels are too long, one of the possibilities is to use abbreviations or numerical references with a legend for explanation.

A similar strategy can be chosen for edge labelling, where the labels are located near the center of the edge. For edges that are predominantly vertical, markings should be placed to the left or right of the edge. Conversely, for horizontal edges, the markings are located above or below the edges. However, a convention should be followed that places all markings either left / up or right / down from the edge. Otherwise, the label may be misassigned to the edge.

With a large number of different labels to be displayed, or with an excessive length, it is obvious that the simultaneous display of all labels is inefficient. We can use one of the existing strategies to solve this problem.

A common solution is to display the label only in a small region of the graph, for example in an area around the current cursor position. If the display density is too high, various visualization distortions are required (see below) so that we can devote a larger area of the screen space to the given section of the graph. An alternative to this solution may be to rotate the graphs to reduce label overlaps (see figure).



Another interesting solution is to display only a random subset of labels for a short time and then display another random subset, etc. The idea behind this solution is that the observer's short-term memory allows the observer to memorize more labels than using a static display.

Interactions

Although we will get acquainted with the interaction techniques with different visualization environments in detail later, we will mention here several techniques of interaction, which are most often associated with trees and graphs. Some types of interaction, such as camera tracking and zooming, are common to all types of visualization, so we'll mention them here for completeness only. Others, such as **focus+context**, may be applicable to a variety of visualizations, but were originally developed to visualize trees and graphs. That is why we will deal with them in more detail here.

Interaction with a virtual camera

Common interactions, such as panning, zooming, and rotating, are considered simple changes applied to a virtual camera that observes a particular segment of the scene. This allows the user to incrementally observe the entire scene and create a mental model of the objects in the scene and their relationships. Operations of this type are often controlled manually, although there are also automated techniques, such as controlled flyovers over a scene or automatic rotation of 3D objects.

Interaction with graph elements

Most interactions of this type begin with selection, where one or more graph components are isolated. Highlighting, deleting, masking, shifting, or retrieving details is then performed on these components. For example, graphs with cluttered clusters of data can be edited by selecting a set of nodes and dragging them to an area of the screen that is less populated by nodes, while, of course, maintaining their connection to other nodes.

Similarly, we can select and move or change the shape of the edges to eliminate their crossing or to increase the aesthetic value of the graph.

Selections can contain a single object, all objects in a given region or distance, or objects that meet user-defined constraints (for example, all nodes directly connected to a given node). One of the biggest problems in selecting elements in a graph occurs in dense regions, where the individual elements are so close to each other that unambiguous selection is difficult or even impossible. This leads to the need to introduce other types of interaction, such as zooming or deformation techniques, which we will discuss later.

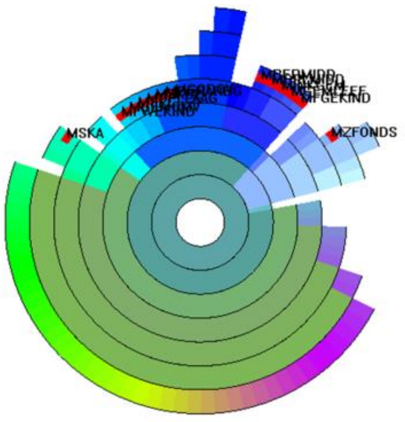
Interaction with graph structure

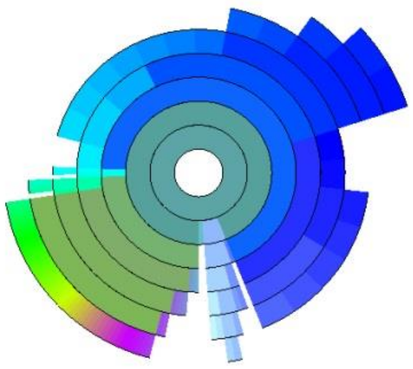
We distinguish two basic classes of interaction, which are focusing directly on graph structures.

- The first class changes the structure of the graph itself. For example, rearranging the order of tree branches can reveal relationships that were not visible in the original layout. Rearranging columns and rows in a matrix visualization can again highlight new properties or relationships between data.

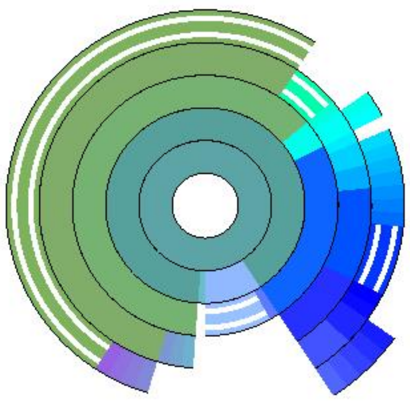
- The second class of interactions associated with the graph structure are the so-called **focus+context** techniques, where a certain subset of a given structure (focus) is presented in detail, while the rest of the structure is displayed only in outlines so that the user has the appropriate context. One of the most popular techniques of this type is fisheye, where the parts of the graph falling within the area of interest are magnified by non-linear scaling, while the parts outside the area of interest are proportionally reduced. This type of deformation can be performed in screen space (derived from pixels) or in structure space (derived from graph components). The second case is suitable for visualizing graphs, where we can, for example, enlarge one branch of a tree at the expense of other branches or we can highlight all the edges emanating from a given node to better detect its neighboring nodes.

The blue area of the graph on the left has been highlighted in the image on the right, making it easier to explore and interactively select in this area.





A technique that can be considered as similar to the two classes of interaction mentioned is the selective hiding or removal of graph sections. For example, if a tree branch has already been explored, the user may want to remove it from the screen to make more room for unexplored regions. In this sense, it can be considered as a change of structure (deletion of a component) or a reduction of details. The figure shows several subtrees that have been hidden and a double white band informs the user that more detailed information is located below these nodes.



Text and document visualization

We currently have a huge number of information sources at our disposal – from libraries to archives to all applications running on the Internet. Visualization is an irreplaceable helper for the analysis of this data. Blog, wiki, billions of words, a set of papers, or a digital library can be viewed in several ways. Because the chosen method of visualization depends on the given task, we will look at what tasks we must deal with when working with text, documents, or objects on the web.

One of the most common operations on texts and documents is searching for a word, phrase, or topic. If the data is at least partially structured, we can also look for relationships between words, phrases, topics, or documents. For structured texts or document files, searching for patterns or outlines in texts or documents is often a key task.

We define a set of documents as a **corpus**. Then we work with objects inside these corpora. These objects can be words, sentences, paragraphs, entire documents, or even other sets of documents. We can also consider pictures and videos as objects. These objects are often considered atomic with respect to a given task, analysis, or visualization.

Texts and documents are often structured and contain several attributes and metadata. For example, documents have a given format and include metadata about the document (e.g., author, creation date, modification date, comments, size, …).

Systems for mining information from documents work based on querying, where as a result we receive the relevant document or its part that corresponds to the query. However, this requires pre-processing of the document and interpretation of the semantics of the text.

We can also calculate statistics about documents, such as the number of words or paragraphs, the distribution of words or their frequency. This way, for example, we can identify the author.

We can also identify relationships between paragraphs or documents in the corpus. For example, we may ask, "Which documents relate to the spread of influenza?" This is a non-trivial query, since searching for word "influenza" is not sufficient.

Similarity can be defined, for example, in the form of citations, co-authorships, topic, etc.

We define three levels of text representation: **lexical**, **syntactic,** and **semantic**. Each stage requires a conversion of unstructured text into some form of structured data.

Lexical level

The lexical level consists in transforming a string of characters into a sequence of atomic entities called **tokens**. Lexical analyzers process a sequence of characters with a given set of rules into a new sequence of characters (tokens), which can be used for subsequent analyses. Tokens can contain characters, n-grams, words, lexemes (vocabulary units), phrases, etc., all with associated attributes. Many types of rules are used to extract tokens, the most common being finite state machines defined by regular expressions.

Syntactic level

The syntactic level deals with the identification and annotation of the functions of each token. In this way we assign different marks, such as the position of a sentence or whether the word is a noun, adjective, complement, conjunction… Tokens can also have attributes such as whether they are singular or plural or their relationship to other tokens. "Richer" tags contain a date, place, person, organization, or time. The process of extracting these annotations is called **NER – named entity recognition**. The richness and great diversity of language models and grammars provide a wide range of approaches.

Semantic level

The semantic level involves the extraction of meaning and relationships between knowledge derived from the structures identified in the syntactic step. The aim of this degree is to define the analytical interpretation of the whole text in a given context or even independently of the context.

Vector Space Model (VSM)

Vector space model is an algebraic model for the representation of text documents (and generally any objects) in the form of vectors of identifiers. It is used for information filtering, information mining, indexing, etc.

Documents are represented by vectors. Each dimension in the vector corresponds to a certain term (words, keywords, or even longer phrases). If a term occurs in a document, its value in the vector is nonzero.

The calculation of so-called "term vectors" is a crucial step in the visualization of documents and corpora and analysis techniques.

In the Vector Space Model, the **term vector** of an object of interest (paragraph, document, set of documents) is a vector in which each dimension represents the weight of a given word in the document. Furthermore, so-called "stop words" (e.g., the, a) are typically filtered out to remove noise, or words with the same base (root) are aggregated.

The following pseudocode counts the occurrences of unique tokens, except for stop words. The token stream generated from one document using a lexical analyzer is taken as input.

The variable labeled as **terms** contains a hash table that maps unique terms to the number of their occurrences in the document.

```
Count-Terms(tokenStream)
```

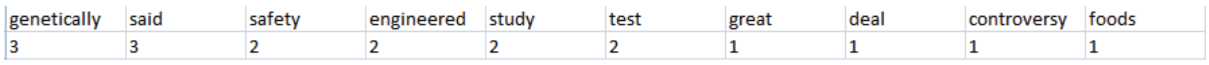
```
terms: = ∅; / * initialize terms to an empty hash table * /
for each token t in tokenStream
  do if t is not a stop word
     do increment (or initialize to 1) terms[t];
return terms;
```

Example:

Sample text:

There is a great deal of controversy about the safety of genetically engineered foods. Advocates of biotechnology often say that the risks are overblown. ‘‘There have been 25,000 trials of genetically modified crops in the world, now, and not a single incident, or anything dangerous in these releases,’’ said a spokesman for Adventa Holdings, a UK biotech firm. During the 2000 presidential campaign, then-candidate George W. Bush said that ‘‘study after study has shown no evidence of danger.’’ And Clinton Administration Agriculture Secretary Dan Glickman said that ‘‘test after rigorous scientific test’’ had proven the safety of genetically engineered products.

- The previous paragraph contains 98 string tokens, 74 terms, and 48 terms after deleting stop words
- Example of term vector generated by pseudocode:



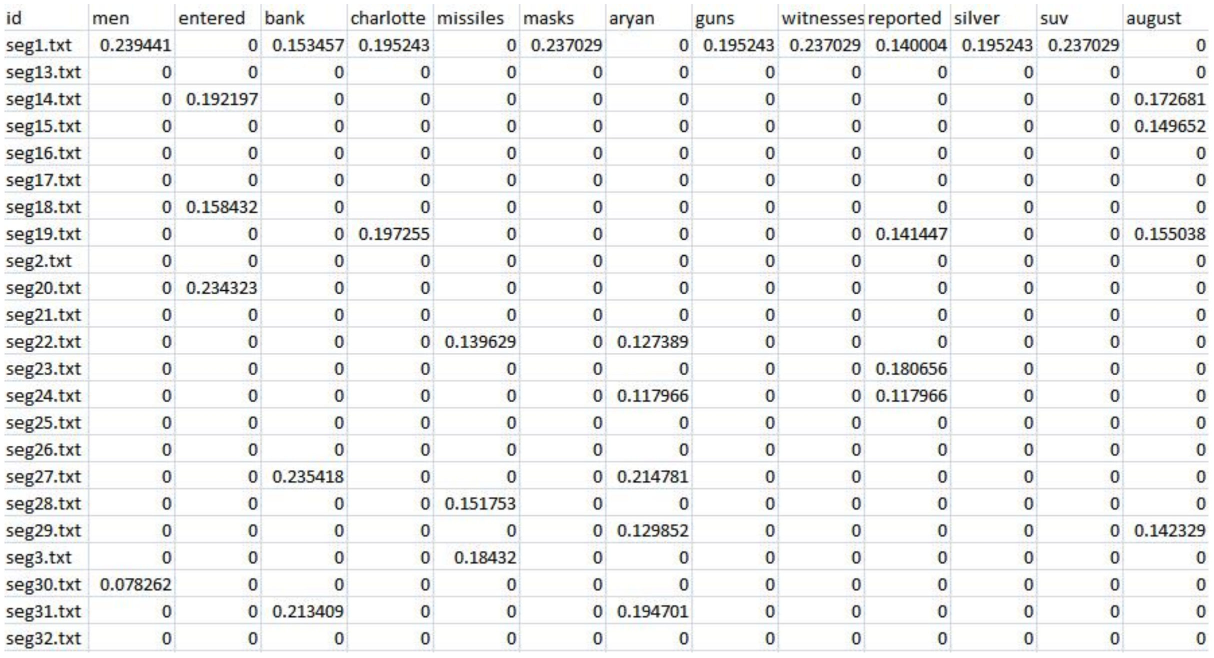
genetically	said	safety	engineered	study	test	great	deal	controversy	foods
3	3	2	2	2	2	1	1	1	1

VSM – counting weights

This vector space model requires a scheme for assigning weights to terms in a document. There are several methods for assigning weights, the best known of which is the "**term frequency inverse document frequency**" – **tf-idf**. Let Tf(w) be the term frequency or the number of occurrences of the word w in a document and Df(w) be the document frequency or the number of documents that contain the word w. Let N be the number of all documents. Then we define TfIdf(w) using a formula

$$TfIdf\,(w) = Tf\,(w) * \log\left(\frac{N}{Df\,(w)}\right)$$

This determines the relative importance of a given word in a document, which corresponds to our intuitive view of the importance of words. We are interested in words that appear frequently in one document but are not as common in other corpus documents. Folowing table shows the term vectors for a group of documents using tf-idf scales.



id	men	entered	bank	charlotte	missiles	masks	aryan	guns	witnesses	reported	silver	suv	august
seg1.txt	0.239441	0	0.153457	0.195243	0	0.237029	0	0.195243	0.237029	0.140004	0.195243	0.237029	0
seg13.txt	0	0	0	0	0	0	0	0	0	0	0	0	0
seg14.txt	0	0.192197	0	0	0	0	0	0	0	0	0	0	0.172681
seg15.txt	0	0	0	0	0	0	0	0	0	0	0	0	0.149652
seg16.txt	0	0	0	0	0	0	0	0	0	0	0	0	0
seg17.txt	0	0	0	0	0	0	0	0	0	0	0	0	0
seg18.txt	0	0.158432	0	0	0	0	0	0	0	0	0	0	0
seg19.txt	0	0	0	0.197255	0	0	0	0	0	0.141447	0	0	0.155038
seg2.txt	0	0	0	0	0	0	0	0	0	0	0	0	0
seg20.txt	0	0.234323	0	0	0	0	0	0	0	0	0	0	0
seg21.txt	0	0	0	0	0	0	0	0	0	0	0	0	0
seg22.txt	0	0	0	0	0.139629	0	0.127389	0	0	0	0	0	0
seg23.txt	0	0	0	0	0	0	0	0	0	0.180656	0	0	0
seg24.txt	0	0	0	0	0	0	0.117966	0	0	0.117966	0	0	0
seg25.txt	0	0	0	0	0	0	0	0	0	0	0	0	0
seg26.txt	0	0	0	0	0	0	0	0	0	0	0	0	0
seg27.txt	0	0	0.235418	0	0	0	0.214781	0	0	0	0	0	0
seg28.txt	0	0	0	0	0.151753	0	0	0	0	0	0	0	0
seg29.txt	0	0	0	0	0	0	0.129852	0	0	0	0	0	0.142329
seg3.txt	0	0	0	0	0.18432	0	0	0	0	0	0	0	0
seg30.txt	0.078262	0	0	0	0	0	0	0	0	0	0	0	0
seg31.txt	0	0	0.213409	0	0	0	0.194701	0	0	0	0	0	0
seg32.txt	0	0	0	0	0	0	0	0	0	0	0	0	0

The pseudocode calculates the tf-idf vectors of each document in a given set of documents. It uses the Count-terms function described in the previous pseudocode. The first part iterates through all documents and calculates and stores the term frequency and document frequency. The second part calculates the tf-idf vectors for each document and stores them in a table.

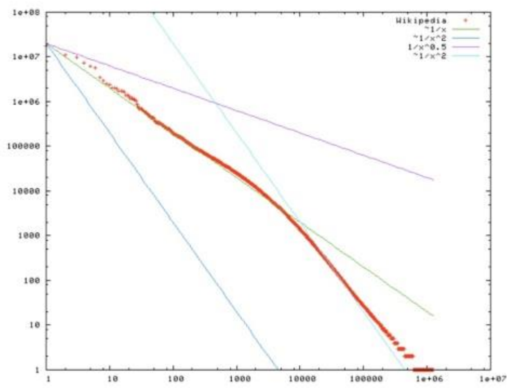
```
Compute-TfIdf(documents)
1    termFrequencies ← Ø  // Looks up term count tables for document names.
2    documentFrequencies← Ø  // Counts the documents in which a term occurs.
3    uniqueTerms← Ø  // The list of all unique terms.
4    for each document d in documents
5              do docName ← Name(d)  // Extract the name of the document.
6      tokenStream ← Tokenize(d)  // Generate document token stream.
7      terms ← Count-Terms(tokenStream)  // Count the term frequencies.
8      termFrequencies[docName] ← terms // Store the term frequencies.
9      for each term t in Keys(terms)
10            do increment (or initialize to 1) documentFrequencies[t]
11           uniqueTerms← uniqueTerms ∪ t
12
13     tfIdfVectorTable ← Ø // Looks up tf-idf vectors for document names.
14     n ← Length(documents)
15     for each document name docName in Keys(termFrequencies)
16             do tfIdfVector ← create zeroed array of length Length(uniqueTerms)
17     terms ← termFrequencies[docName]
18      for each term t in keys(terms)
19             do tf ← terms[t]
20           df ← documentFrequencies[t]
21           tfIdf ← tf * log(n/df)
22           tfIdfVector[index of t in uniqueTerms]← tfIdf
```

```
            tfIdfVectorTable[docName] ← tfIdfVector
    return tfIdfVectorTable
```

Zipf's law

Most often we encounter a normal and uniform type of value distribution. Economist Vilfredo Pareto argued that the company's return is inversely proportional to its position – a classic power law, which results in the well-known 80-20 rule, where 20% of the population owns 80% of the world's wealth.

Harvard linguist George Kingsley Zipf called the distribution of words in natural language corpora using the discrete power distribution the Zipf distribution. Zipf's law states that in a typical natural language document, the frequency of any word is inversely proportional to its category (rank) in the frequency table. The plot of the Zipf curve on a logarithmic scale in both axes has the shape of a straight line with a slope of -1.



One of the immediate implications of this law is the fact that a small number of words describes most of the key concepts in small documents.

Tasks using Vector Space Model

Vector space model supplemented with some distance metrics allows you to perform a variety of tasks. We can use tf-idf in combination with a vector space model to identify interesting documents. We can answer questions such as: Which documents are similar to a given document? Which documents are relevant to a given set of documents? Which documents are most relevant to a given search query? These are the documents whose term vectors are as similar as possible to the specified document, an average term vector through the document set, and a search query vector.

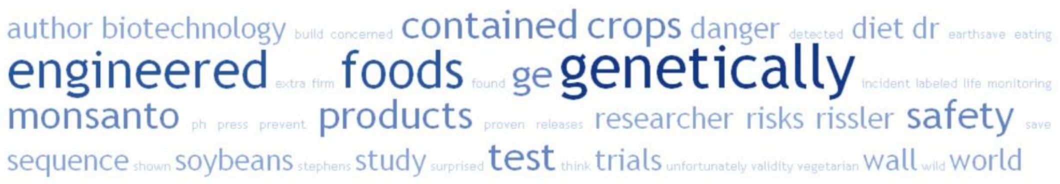
Another indirect task may be to provide the user with the meaning of the whole corpus. The user can search for patterns or clusters and theme distribution through a set of documents. This often involves visualizing the corpus in a 2D layout or presenting it to the user in the form of a graph with links between documents for better navigation between them.

The visualization pipeline maps well to document visualization: we obtain data (corpus), transform it into vectors, then run the appropriate algorithms based on the given task (similarity, search, clustering…) and generate the visualization.

Visualization of individual documents

We will now look at several techniques for visualizing a single document.

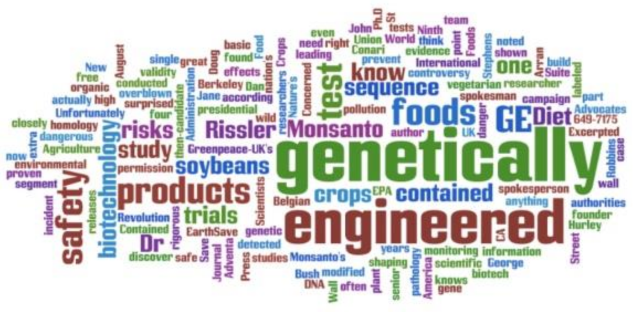
Tag clouds – also known as text clouds or word clouds, are layouts of individual tokens that are colored, and their size is set according to their frequency in the document.



author biotechnology build concerned contained crops danger detected diet dr earthsave eating
engineered extra firm foods found ge genetically incident labeled life monitoring
monsanto ph press prevent products proven releases researcher risks rissler safety save
sequence shown soybeans stephens study surprised test think trials unfortunately validity vegetarian wall wild world

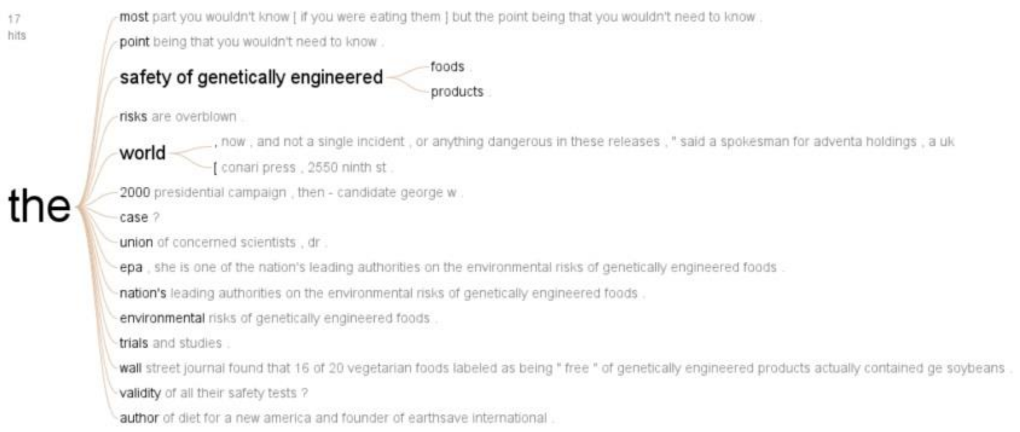
Font size and hue are proportional to the frequency of words in the document.

Text clouds and their variations (such as Wordle – see image below) are examples of visualization that uses only the frequency of term vectors and layer layout algorithms.



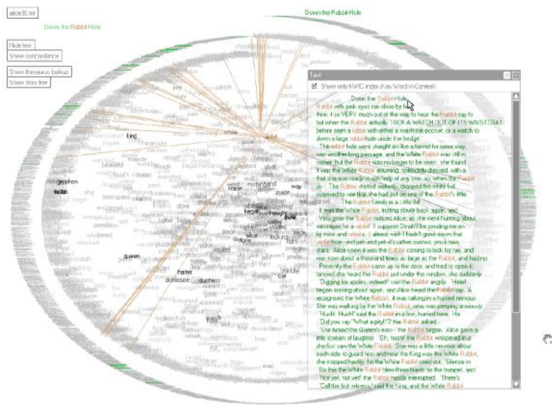
Font size corresponds to the frequency of words in the document.

WordTree is a visual representation of both the frequency of terms and their context (see picture). Size is used to represent the frequency of a term or phrase. The root of the tree is a user-defined word or phrase, and the branches represent the various contexts in which the word or phrase is used in the document.

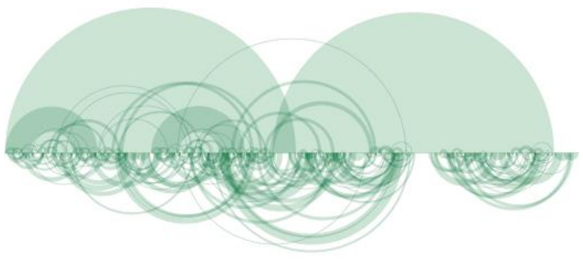

17 hits
the
most part you wouldn't know [if you were eating them] but the point being that you wouldn't need to know .
point being that you wouldn't need to know .
safety of genetically engineered
foods .
products .
risks are overblown .
world
, now , and not a single incident , or anything dangerous in these releases , " said a spokesman for adventa holdings , a uk
[conari press , 2550 ninth st .
2000 presidential campaign , then - candidate george w .
case ?
union of concerned scientists , dr .
epa , she is one of the nation's leading authorities on the environmental risks of genetically engineered foods .
nation's leading authorities on the environmental risks of genetically engineered foods .
environmental risks of genetically engineered foods .
trials and studies .
wall street journal found that 16 of 20 vegetarian foods labeled as being " free " of genetically engineered products actually contained ge soybeans .
validity of all their safety tests ?
author of diet for a new america and founder of earthsave international .

TextArc

We can extend the representation of word layouts with connectivity visualization. There are several ways to calculate connectivity. TextArc is a visual representation of how terms are related to the lines of text in which they appear (see figure). Each word of the text is drawn around an ellipse. As with Text Clouds, words with a higher frequency are rendered larger and brighter. Higher frequency words are drawn inside the ellipse depending on their occurrence on the circle (similar to the RadViz technique).



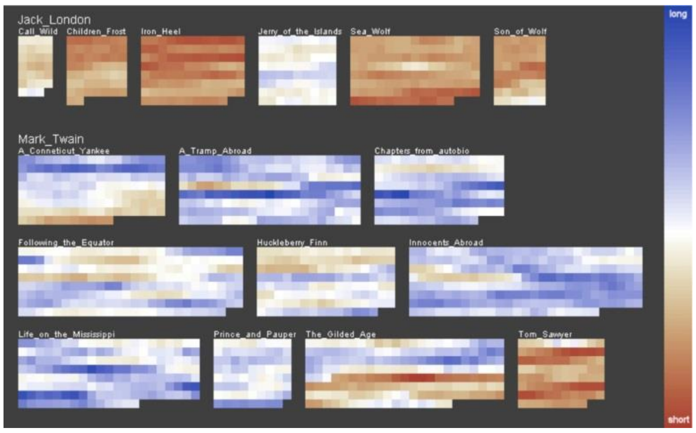
Arc Diagrams is a technique aimed at displaying repetitions in text or any sequence. Repeating subsequences are identified and connected by semicircular arcs. The thickness of the arc represents the length of the subsequence, the height of the arc represents the distance between the subsequences.



The picture visualizes Bach's Minuet in G major, where the classical structure of the Minuet can be seen. It contains 2 parts, each consisting of a long passage that is played twice. The parts are not too related. The overlap of these two parts shows that the end of the first part is the same as the beginning of the second part.

Literature fingerprinting is a method of displaying the properties that characterize a given text. Instead of counting the values of only one of the properties or the vector representing the text, we compute a sequence of property values for the whole text and present it to the user as a characteristic "fingerprint" of the document. This allows the user to investigate the document and analyze the evolution of values throughout. In addition, the structural information about the document is used to visualize the document at various levels of resolution.

This method is used, for example, to verify authorship.



The figure shows an example of authorship verification. Each pixel represents a block of text, and the pixels are clustered into books. The color is mapped here to the average length of sentences. It can be seen that the fingerprints of Jack London's and Mark Twain's books are visually different.

Document set visualization

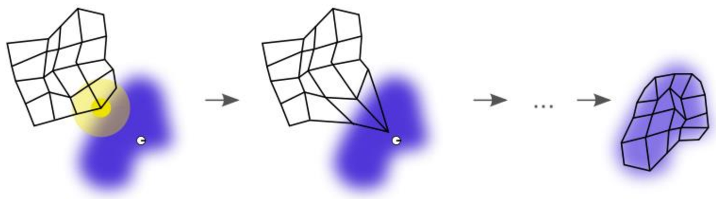
When visualizing a set of documents, the goal is usually to place similar documents close to each other, and, conversely, the dissimilar ones as far apart as possible. This is a problem of finding the minimum and maximum values – $O(n^2)$. We calculate the similarity between all pairs of documents and thus determine their distribution. Common approaches are graph spring layouts, multidimensional scaling, clustering, and self-organizing maps.

A **self-organizing map (SOM)** is created with a machine learning algorithm using a collection of typically 2D nodes into which we place documents. Each node has an associated vector of the same dimensionality as the input vectors (document vectors) that were used to train the map.

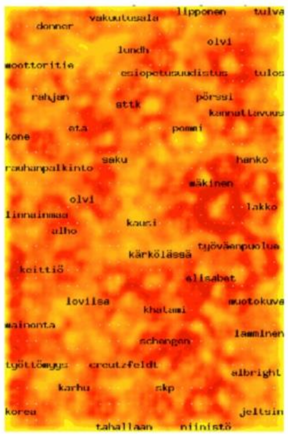
SOM is thus a type of artificial neural network that creates a low-dimensional (typically 2D) discrete representation of the sample input space, which we call a map. To preserve the topology of the input space, a function that considers neighbors, is used.

At the beginning, SOM nodes are initialized – typically random values are set for them. Then a random vector is selected from the set of input vectors and its distance from all other nodes is calculated. We assign weights to the nearest nodes (in a given radius) – the nearest node has the largest weight. When iterating through input vectors, the radius gradually decreases.

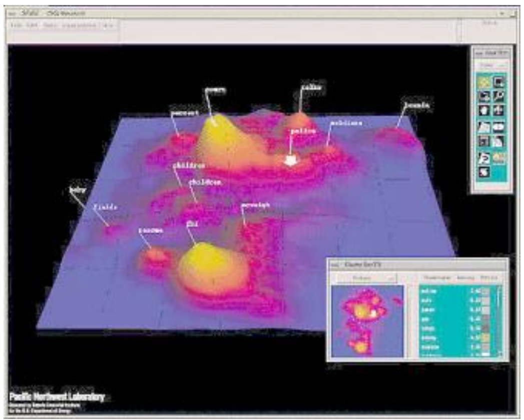
The training of the neural network with respect to neighbors works as follows. The blue cloud represents the distribution of the training data, and the small white circle indicates the current training sample. Initially, the SOM nodes are arranged arbitrarily in the data space. The node closest to the current training sample (yellow) is moved to that sample. After a few iterations, we get to the result on the right.



An example of the use of SOM for text data is shown in the figure, which shows a million documents collected from 83 discussion groups in Finland. Labels indicate areas of discussion topics; color represents the number of documents – lighter areas mean higher occurrence.

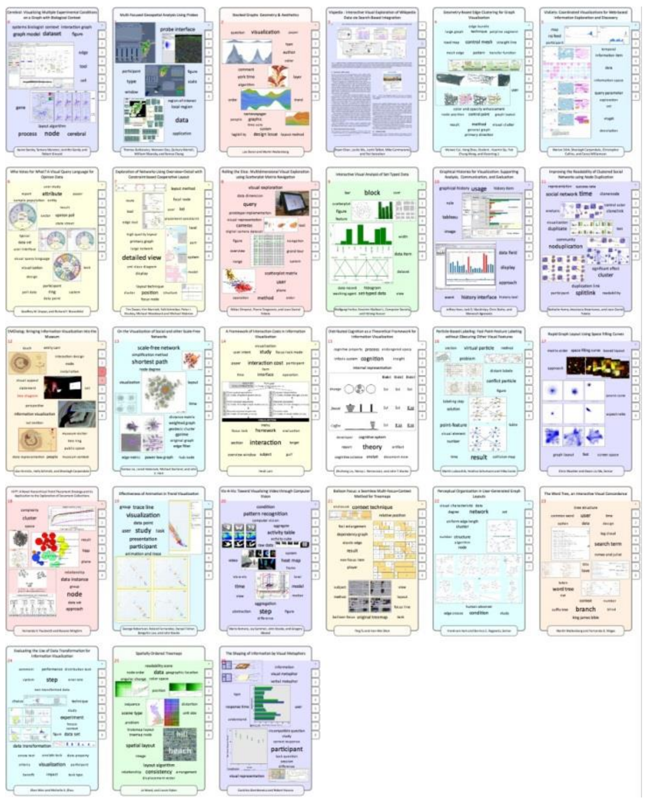


Themescapes are basically summary information about corpora in the form of 3D landscapes, where height and color are used to represent the density of similar documents.



Document cards are compact visualizations representing the key semantics of documents in the form of a mixture of images and key terms, which are obtained using advanced text mining algorithms (based on automatic extraction of the document structure). Images and their captions are derived using graphical heuristics.

In addition, a color histogram is used for the images, that are sorted and classified into individual classes (class 1: photographs / rendered images, class 2: diagrams / sketches / graphs, class 3: tables) and then at least one representative of each non-empty class is displayed.



The figure shows the IEEE Infovis 2008 proceedings corpus, which is represented by a matrix of document cards. The term frequency on each page is highlighted on the right side of the card (the redder, the higher the frequency – e.g., the left document in row 3).

Advanced methods of text visualization

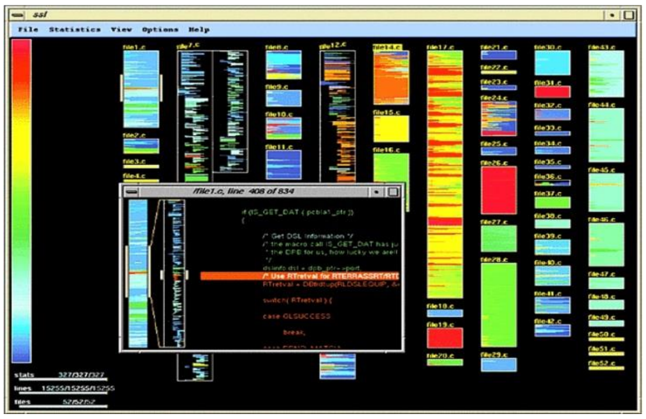
We will now focus on several visualization techniques for texts that include metadata.

- Software Visualization
- Search Result Visualization

- Temporal Document Collection Visualizations
- Representing Relationships

Software Visualization

Eick et al. developed a visualization tool called SeeSoft, which we have already encountered, and which visualizes statistics for each line of code (e.g., "age" and number of modifications, programmer's name, date). In the figure, each column represents a source code file, where the height of the column corresponds to the size of the file. If the file is larger than the screen height, the next column continues. Each line in the columns represents one line of code. If the source file is too large, each line of code is represented by a pixel in the row in the column. This significantly increases the number of rows we can display. The color is mapped to the number of calls. The redder the line, the more often it is called. Conversely, the lines called rarely are blue. The color can also be used for other parameters, such as the time of the last modification or the number of modifications.



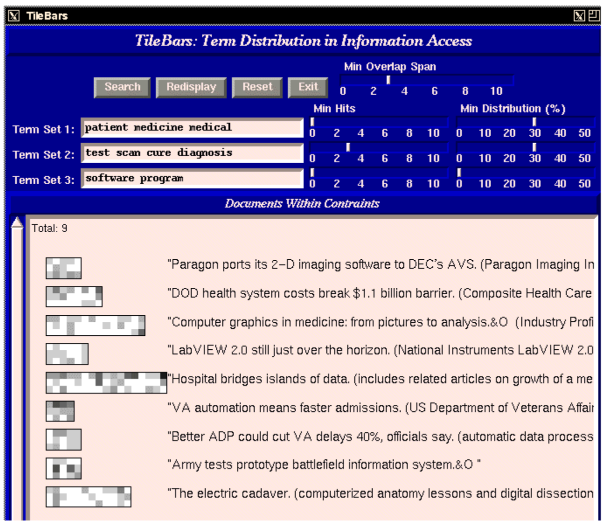
```
if (IS_GET_DAT ( pcblat_ptr ))
{
    /* Get DSL information */
    /* the macro call IS_GET_DAT has pu
     * the DPB for us, how lucky we are!!
     */
    dslinfo.dsl = dpb_ptr->port;
    /* Use RTretval for RTERRASSRT/RTE
    RTretval = DBtidtup(RLDSLEQUIP, &
    switch( RTretval ) {
    case OLSUCCESS
        break;
```

With a window size of 100x100 pixels, SeeSoft can display more than 50,000 lines of code. Screenshot contains 52 files with 15,255 lines of code.

Search Result Visualization

Marti Hearst has developed a simple visualization for query results that is in principle very similar to Keim's pixel displays. He called it TileBars. It displays a variety of statistics related to terms, such as the frequency and distribution of terms, document length, and more.

Each document in the resulting set is represented by a rectangle, where the width indicates the relative length of the document and the layered squares inside correspond to the text segments. Each row of a given stack containing text segments represents a set of query terms, and the shade of the squares indicates their frequency. The titles and first words of the document appear next to their TileBar.



TileBars

TileBars: Term Distribution in Information Access

Search | Redisplay | Reset | Exit

Min Overlap Span: 0 2 4 6 8 10

		Min Hits	Min Distribution (%)
Term Set 1:	patient medicine medical	0 2 4 6 8 10	0 10 20 30 40 50
Term Set 2:	test scan cure diagnosis	0 2 4 6 8 10	0 10 20 30 40 50
Term Set 3:	software program	0 2 4 6 8 10	0 10 20 30 40 50

Documents Within Contraints

Total: 9

"Paragon ports its 2–D imaging software to DEC's AVS. (Paragon Imaging In

"DOD health system costs break $1.1 billion barrier. (Composite Health Care

"Computer graphics in medicine: from pictures to analysis.&O (Industry Profi

"LabVIEW 2.0 still just over the horizon. (National Instruments LabVIEW 2.0

"Hospital bridges islands of data. (includes related articles on growth of a me

"VA automation means faster admissions. (US Department of Veterans Affair

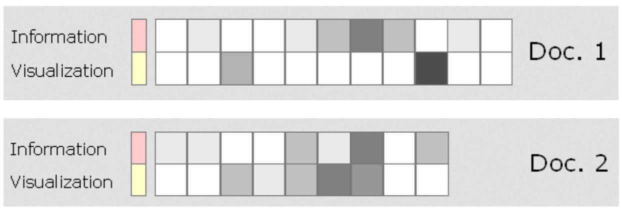
"Better ADP could cut VA delays 40%, officials say. (automatic data process

"Army tests prototype battlefield information system.&O "

"The electric cadaver. (computerized anatomy lessons and digital dissection

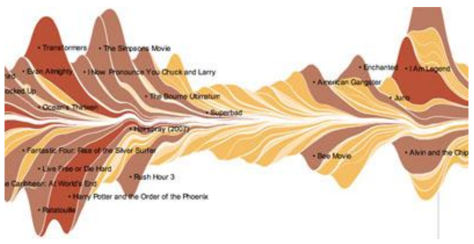
Each large rectangle represents one document, and each small square within the document represents a text segment. The darker the square, the higher the frequency of the set of query terms. This produces a very compact representation and provides information about the structure of the document reflecting the relative length of the document, the frequency of query terms, as well as their distribution.

An example below shows searching for the words Information and Visualization in two different documents. Document 1 is longer, and the individual squares and their color (shade) indicate the frequency of occurrence of the word in the corresponding part of the text. It can be seen that in the first document, these two words do not appear in any part at the same time. On the contrary, in the second, shorter, document, these words appear in three parts at the same time. Therefore, if we are looking for the phrase Information Visualization, the second document will be more suitable for us.

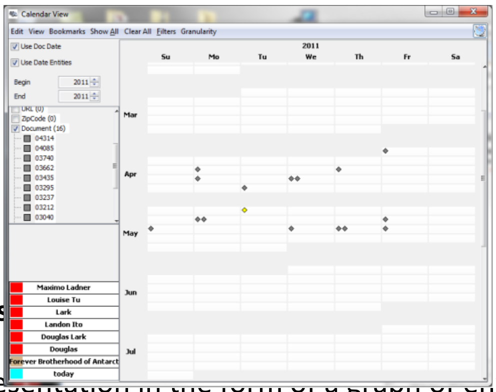


Temporal Document Collection Visualizations

ThemeRiver, also called stream graph, is used to visualize thematic changes over time for a set of documents. This visualization assumes that the input data changes over time. Thematic changes are visually represented by colored horizontal bands, the vertical thickness of which at a given horizontal location represents their frequency at a given point in time.

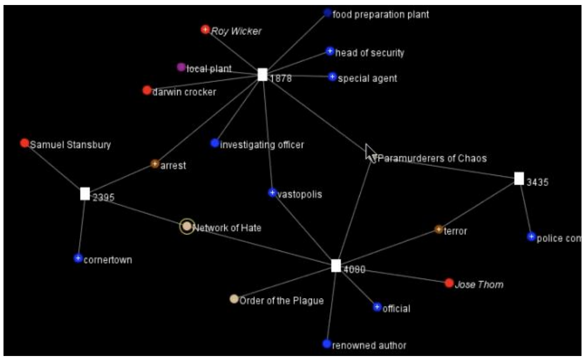


Jigsaw is a tool for visualizing and exploring text corpora. It uses the appearance of a calendar, where document objects are placed in the calendar based on the date when the given entities were identified in the text. When a user clicks on a document, the entities appearing in that document appear immediately.

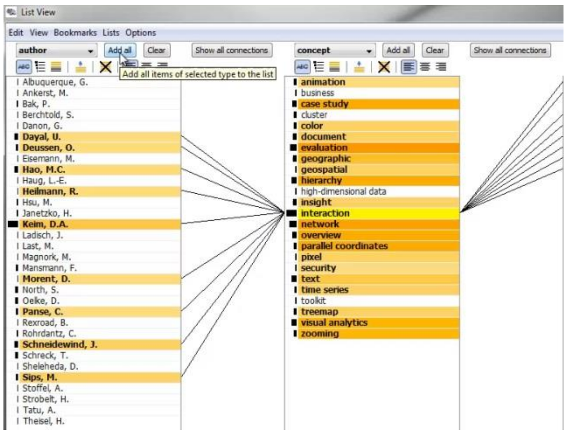


Representing Relationships

Jigsaw also contains a repre tities. Entities are connected with the documents in which they occur. This representation does not show the entities of the entire document collection, but the user can interactively change the areas of interest.



Jigsaw List view is an alternative to the graph view. If the user selects items that interest him, the list view draws connection lines showing their relationships.



Add all items of selected type to the list

Albuquerque, G.
Ankerst, M.
Bak, P.
Berchtold, S.
Danon, G.
Dayal, U.
Deussen, O.
Eisemann, M.
Hao, M.C.
Haug, L.-E.
Heilmann, R.
Hsu, M.
Janetzko, H.
Keim, D.A.
Ladisch, J.
Last, M.
Magnork, M.
Mansmann, F.
Morent, D.
North, S.
Oelke, D.
Panse, C.
Rexroad, B.
Rohrdantz, C.
Schneidewind, J.
Schreck, T.
Sheleheda, D.
Sips, M.
Stoffel, A.
Strobelt, H.
Tatu, A.
Theisel, H.

animation
business
case study
cluster
color
document
evaluation
geographic
geospatial
hierarchy
high-dimensional data
insight
interaction
network
overview
parallel coordinates
pixel
security
text
time series
toolkit
treemap
visual analytics
zooming