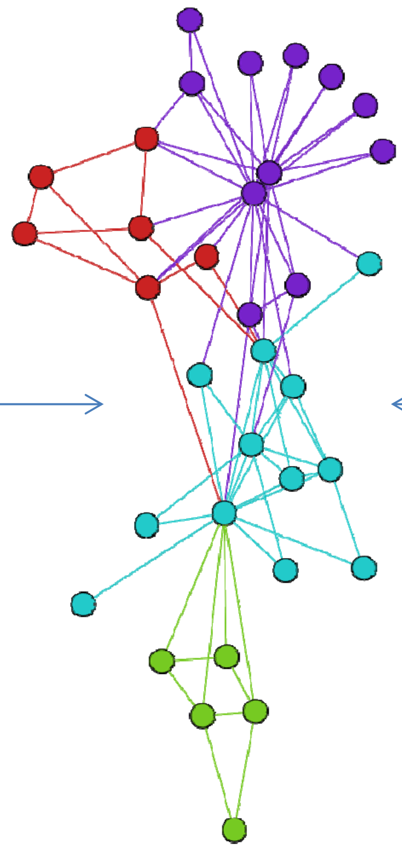
Graph Visualization

What is a Graph?

Relationships between concepts

Mathematics and Graph Theory

↓

Graph Drawing →  ← Network Visualisation

↑

Graph

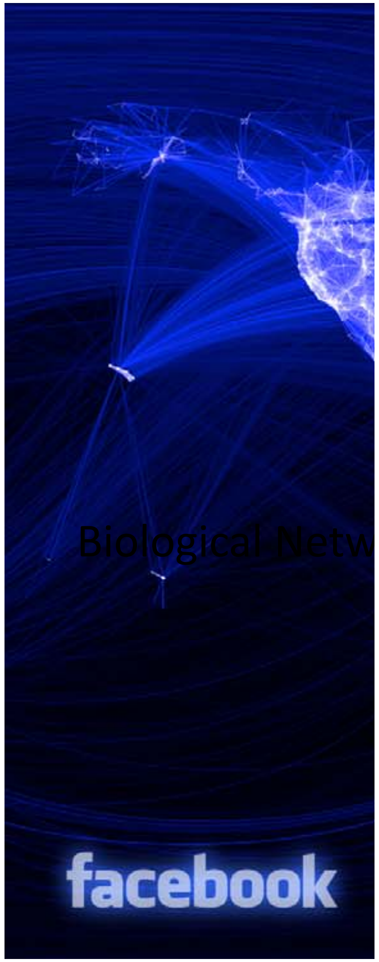
Information Visualisation

↓

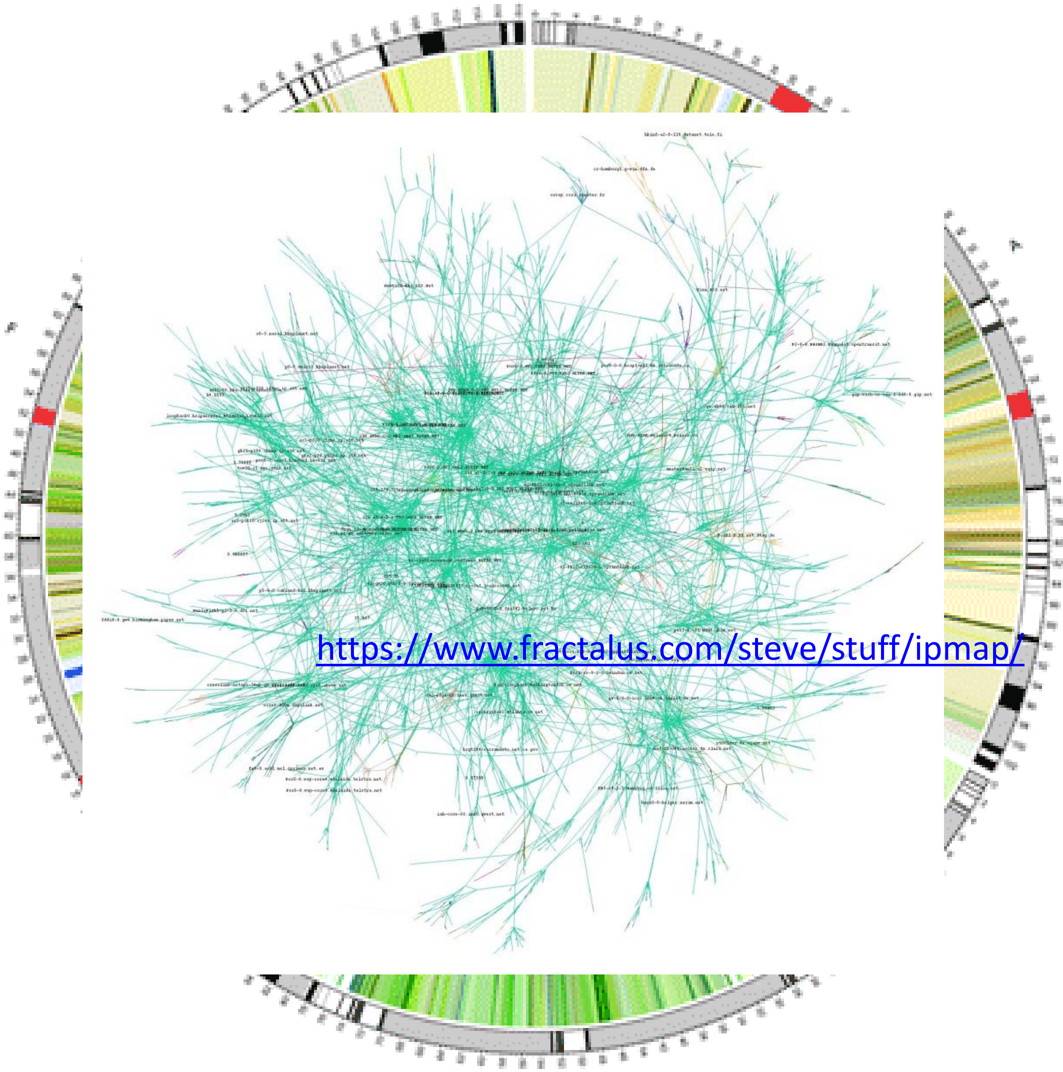
Network Visualisation

↑

Network



Biological Netw



https://www.fractalus.com/steve/stuff/ipmap/

orld Wide Web

map-gallery.html

IP Addresses

http://visualcomplexity.com

Graph Analytics

- **Visualization and statistics are the two basic toolkits one can use on graphs**
- **Complex questions are asked when studying graphs**

- **Easy**
  - Min, max, average, quartiles
  - Exact queries, search

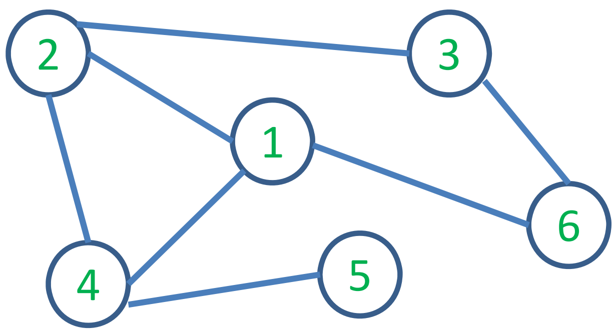
⇨ Excel can do this!

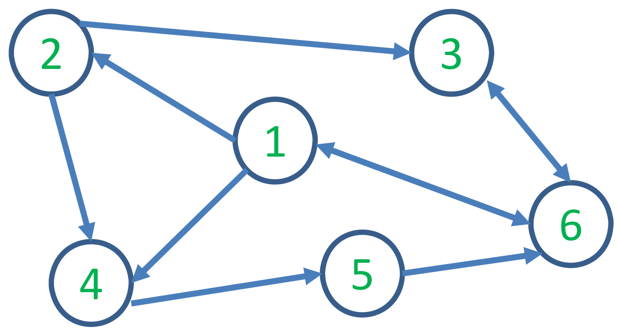
- **Harder**
  - Patterns, trends, correlations
  - Changes over time, context
  - Anomalies, data errors
  - Geographical representation

⇨ Visualization can do this!

What is a Graph

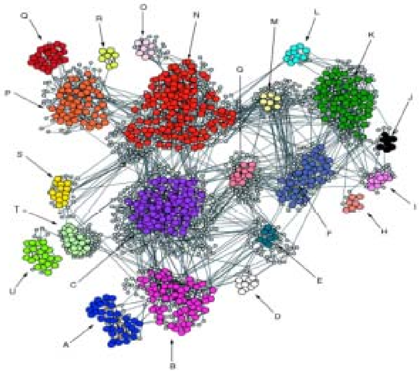
- Graphs, denoted as $G = (V, E)$, are structures formed by a set of vertices, V (also called nodes) and a set of edges, $E = \{v, w\}$, that are connections between pairs of vertices.



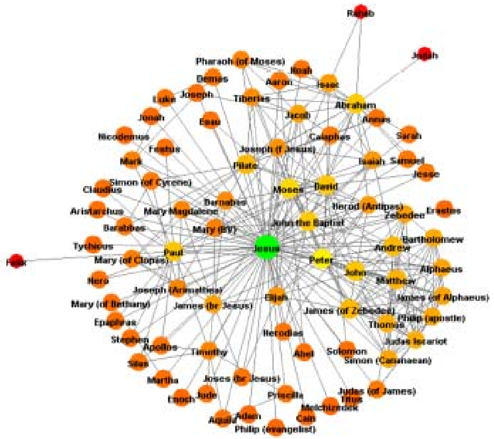


Graphs are everywhere

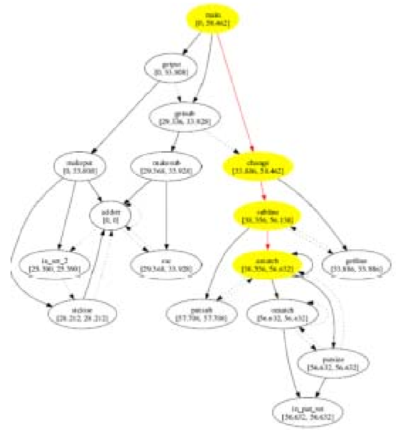
Magwene *et al.* *Genome Biology* 2004 **5**:R100



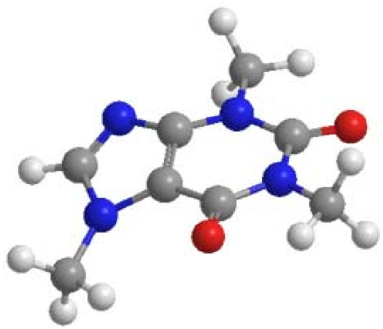
Co-expression Network



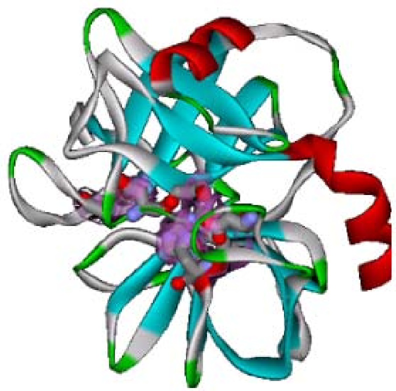
Social Network



Program Flow



Chemical Compound



Protein Structure

Basic Concepts

– The **order** of the graph G, $n = |V|$

– The **size** of the graph G, $m = |E|$

– A graph is planar if it can be drawn in a plane without any of the edges crossing

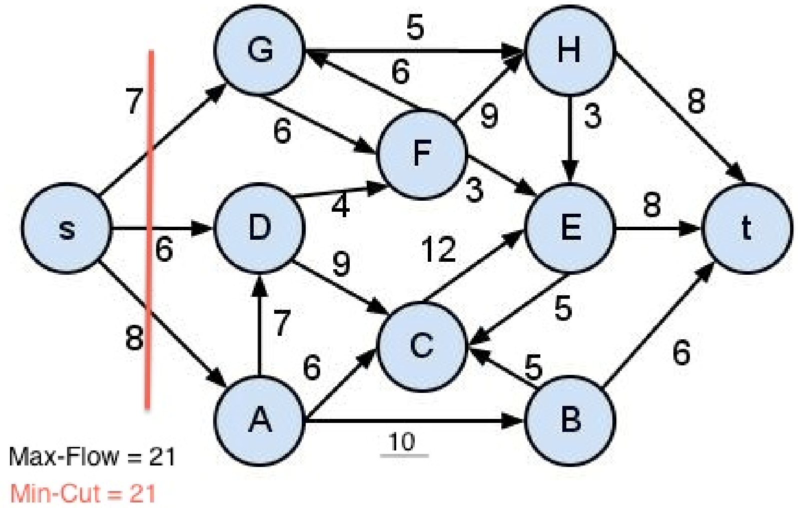


Image source: http://people.seas.harvard.edu/~joshlee/

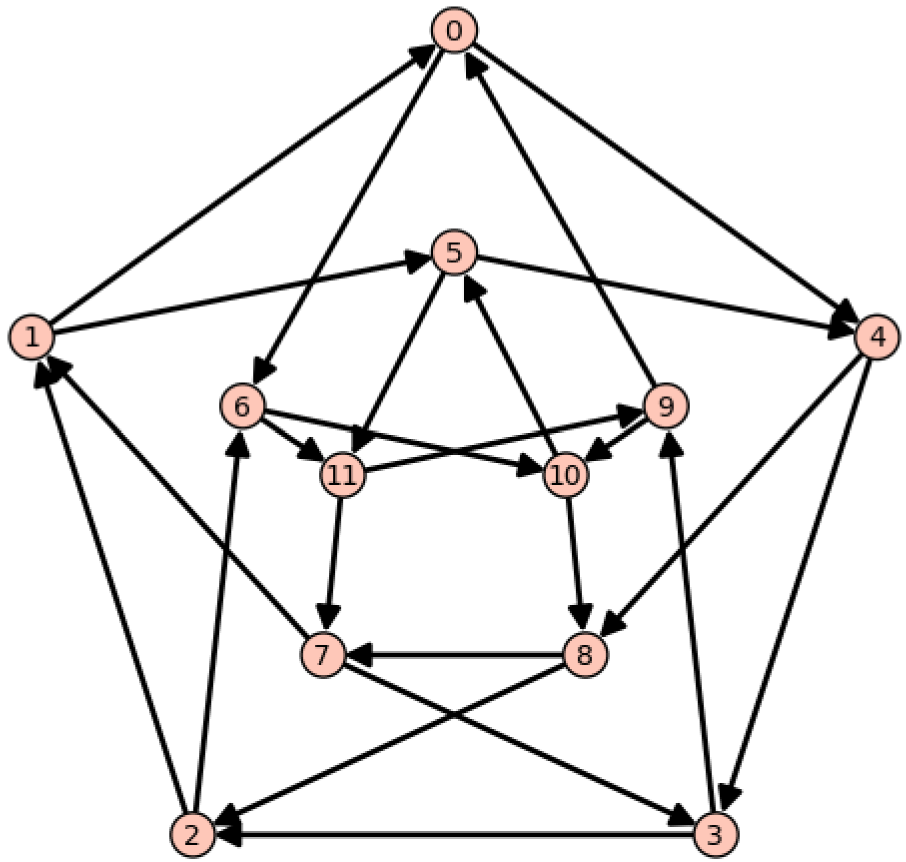
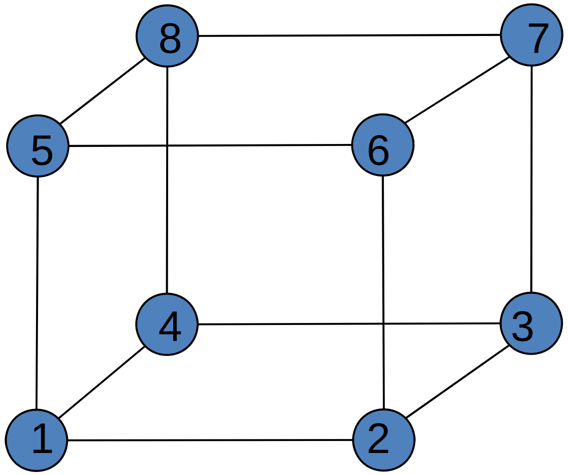
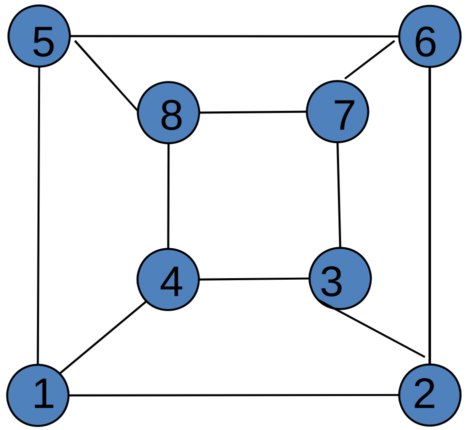


Image source: http://www.sagemath.org/doc/thematic_tutorials/linear_programming.html/



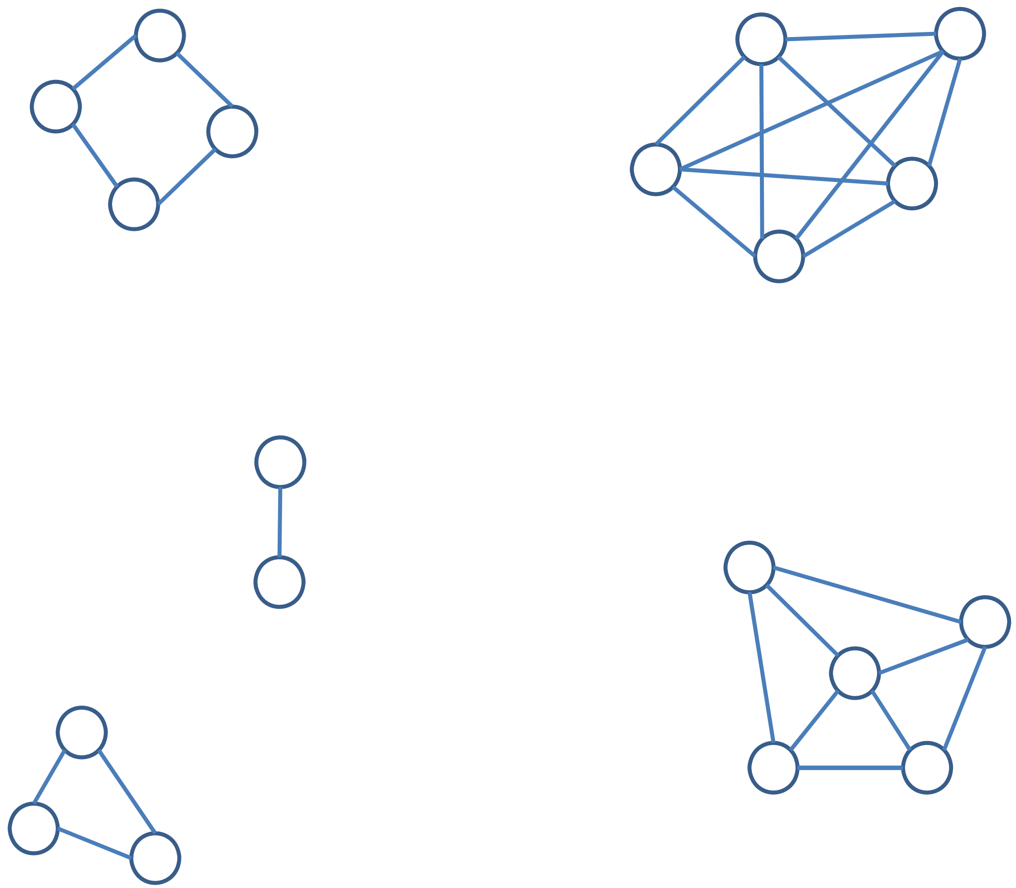
3-D



planar

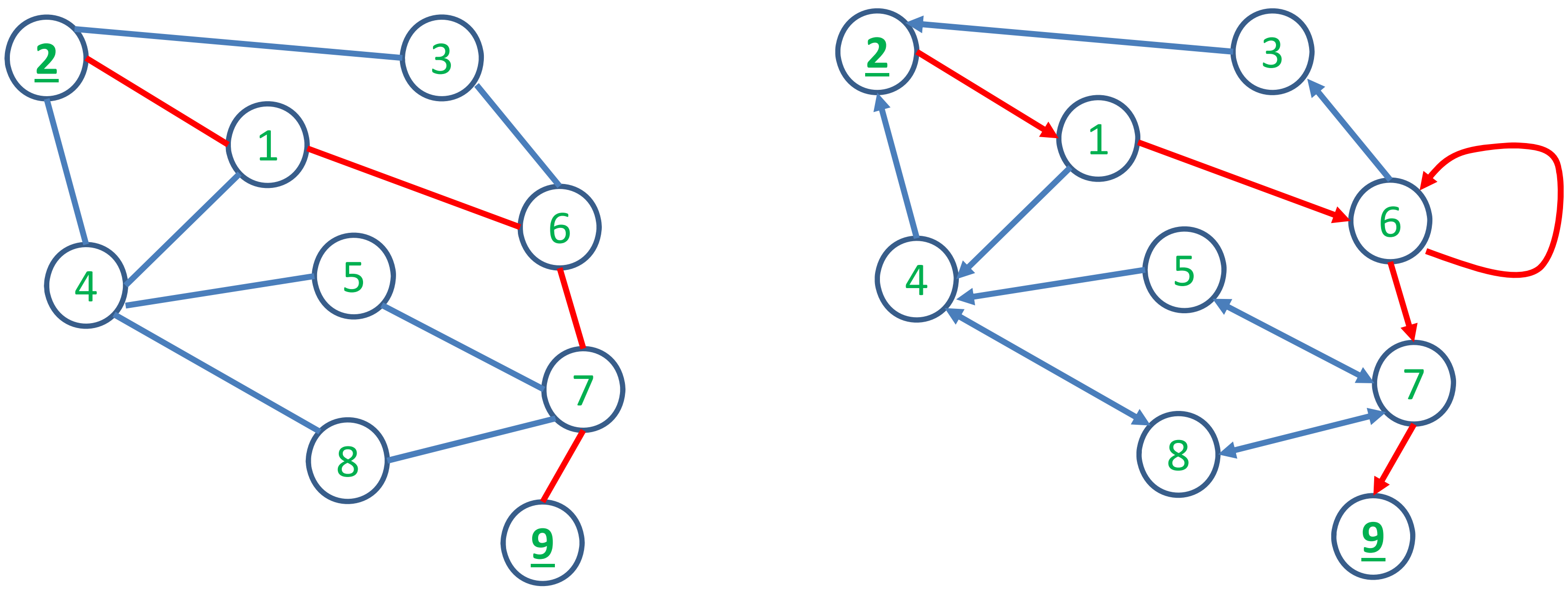
Basic Concepts

- The order of the graph G, $n = |V|$
- The size of the graph G, $m = |E|$
- A graph is planar if it can be drawn in a plane without any of the edges crossing
- The **degree** of a node, $\deg(v)$, is the number of edges that connect to the node
- The density of the graph G, $\frac{m}{\binom{n}{2}}$
- A graph of density 1 is called complete



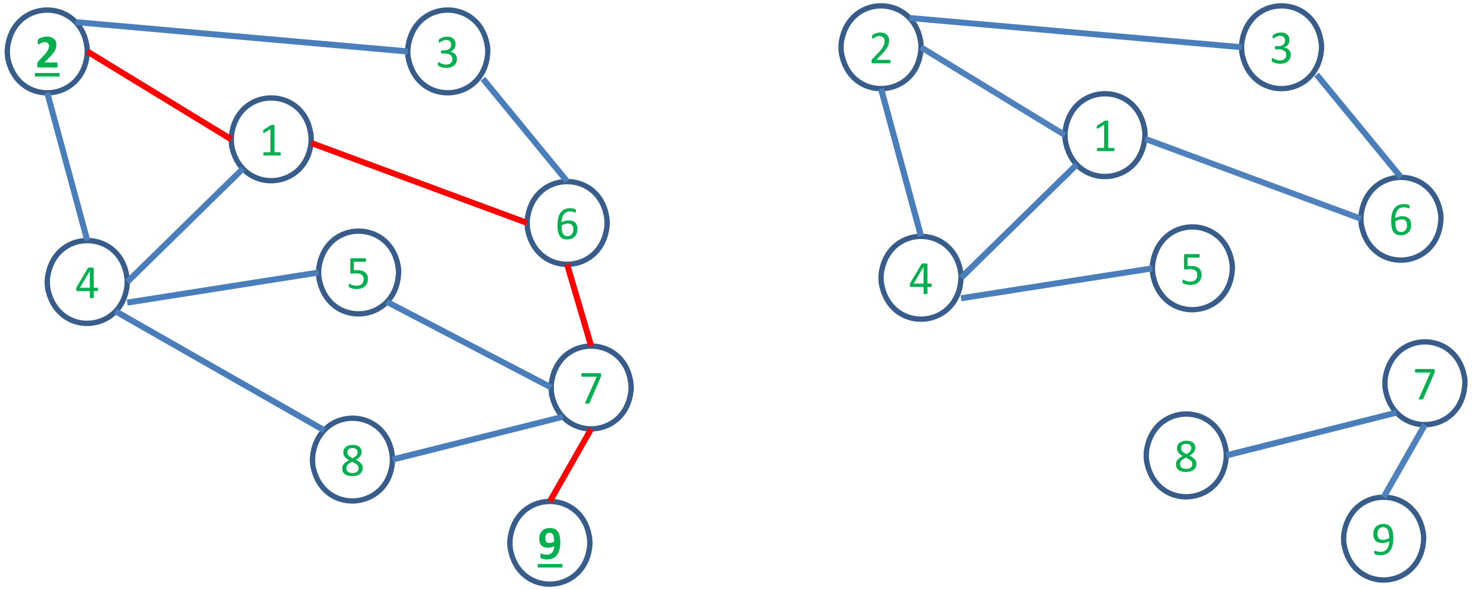
Basic Concepts (II)

- A path from v to u in a graph $G = (V, E)$ is a sequence of edges in E starting at vertex $v_0 = v$ and ending at vertex $v_{k+1} = u$.
- The path is simple if no vertex is repeated



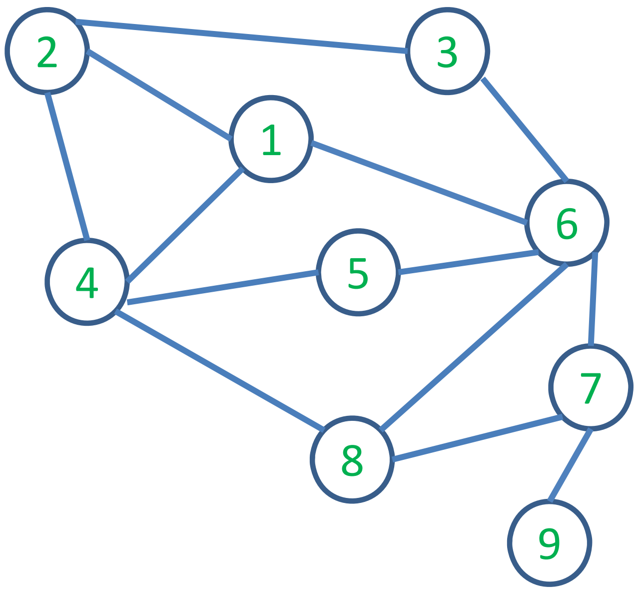
Basic Concepts (II)

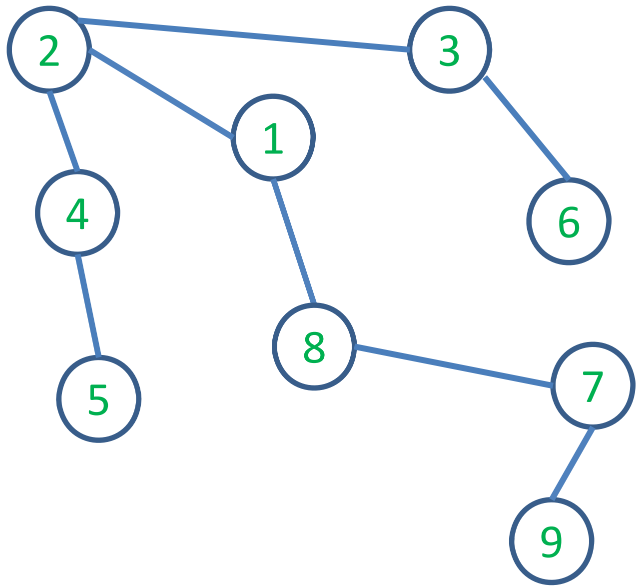
- The length of the path is the number of edges on it
- The distance between two nodes is the shortest path connecting them.
- A graph is *connected* if there exist paths between all pairs of vertices; otherwise, it is *disconnected*.
- The *minimum* number of edges that would need to be removed from G in order to make the graph disconnected is the *edge-connectivity* of the graph.



Basic Concepts (III)

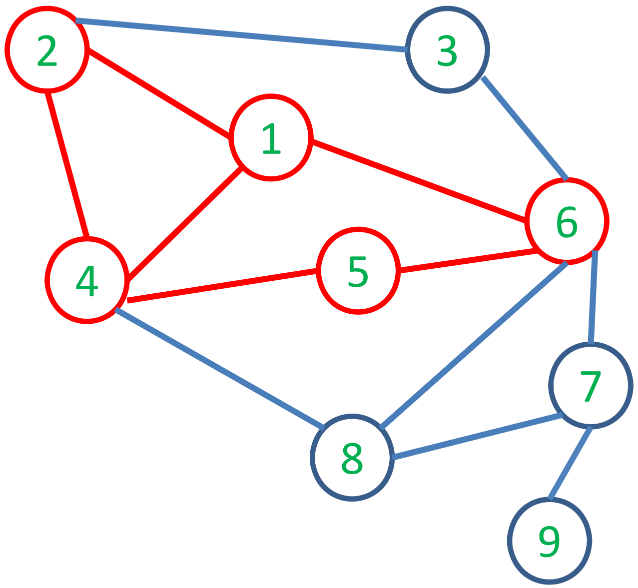
- A <u>*cycle*</u> is a simple path that begins and ends at the same vertex.
- A graph that contains on cycle is <u>*acyclic*</u> and is also called <u>*forest*</u>.
- A connected forest is called a <u>*tree*</u>.





Basic Concepts (IV)

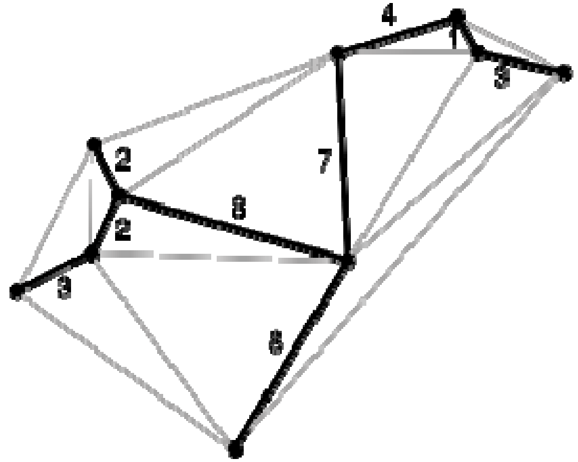
– A *subgraph* $G_s = (S, E_S)$ of $G = (V, E)$ is composed of a set of vertices $S \subseteq V$ and a set of edges $E_S \subseteq E$. G is then a *supergraph* of G_s.



Basic Concepts (IV)

– A connected acyclic subgraph that includes all vertices in V is called a *spanning tree* of G.

- A spanning tree has exactly $n - 1$ edges
- If the edges have weights, the spanning tree with smallest total weights is called the *minimum spanning tree* (there may exist several of them)



http://www.i-cherubini.it/mauro/blog/2006/04/06/minimum-spanning-tree-of-urban-tapestries-messages/

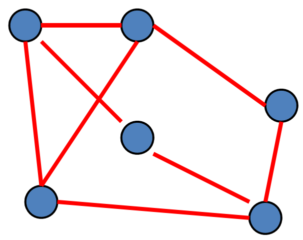
Challenges

- Graph layout
- Scale
- Navigation

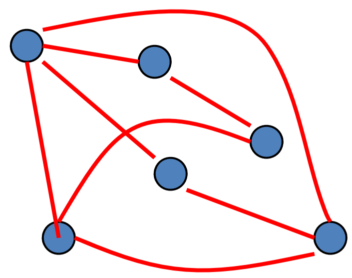
- Problem: Bowl of spaghetti!

Graph Layout Styles

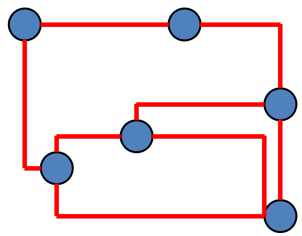
- Straight line



- Arc



- Orthogonal

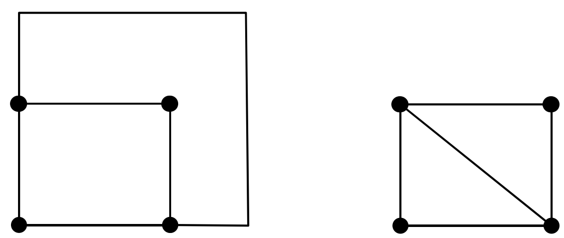


Graph Drawing Aesthetics

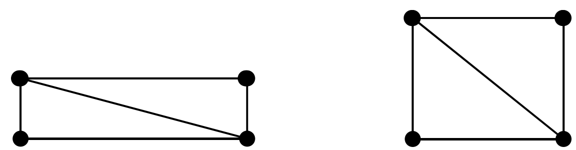
Aesthetics are the graphic properties layout algorithm try to optimise.

- **Crossings**:
  - Minimization of the total crossing number

- **Area**
  - Minimization of drawing area
  - Only meaningful to some layout. Example, grid drawing with integer coordinates



- **Aspect ratio**
  - The ratio of the long and short edge length of its covering rectangle
  - Ideal case is to obtain any aspect ratio in a given range (so the drawing can fit into differently shaped screen space)

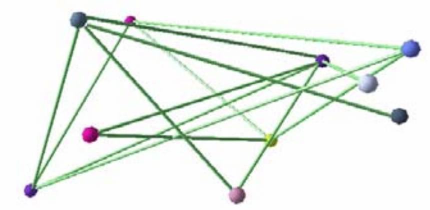


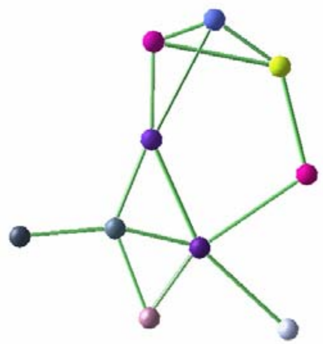
Graph Drawing Aesthetics

- **Edge length (several variations)**:
  - minimization of the sum of the edge length;
  - minimization of the maximum edge length;
  - minimization of the variance of the edge length;
  - only meaningful to some layout algorithm.
- **Bends (several variations):**
  - minimization of the total number of bends;
  - minimization of maximum number of bends on an edge;
  - minimization of the variance of the number of bends on the edge;
  - trivially satisfied by straight-line drawing.

Graph Drawing Aesthetics

- **Angular resolution**:
  - maximization of the smallest angle;
  - especially relevant for straight-line drawing.
- **Symmetry**:
  - display the symmetries of the graph in drawing
  - reflective and rotational symmetry
- **Orthogonality**:
  - how well the edges are parallel to the axes, and how well the nodes match to a grid;
- **Upward flow**:
  - for directed graph only,
  - how well edges are pointing to a specified direction (usually upward);





Basic Graph Layout Techniques

- Force-direct layout
- Adjacent matrix
- Arc-diagram
- Circular layout

Force-Direct Layout of Graph

- We already know:
  - The most common graphical representation of a network is a node-link diagram, where each node is shown as a point, circle, polygon, or some other small graphical object, and each edge is shown as a line segment or curve connecting two nodes.

- **Force-Direct Layout** idea:
  - We imagine the nodes as physical particles that are initialized with random positions, but are gradually displaced under the effect of various forces, until they arrive at a final position. The forces are defined by the chosen algorithm, and typically seek to position adjacent nodes near each other, but not too near.

Force-Direct Layout of Graph

- Specifically, imagine that we simulate two forces: a repulsive force between all pairs of nodes, and a spring force between all pairs of adjacent nodes.

- Let d be the current distance between two nodes, and define the repulsive force between them to be

$$Fr = Kr / d^2$$

(a definition inspired by inverse-square laws such as Coulomb's law), where Kr is some constant.

- If the nodes are adjacent, let the spring force between them be

$$Fs = Ks\,(d - L)$$

(inspired by Hooke's law), where Ks is the spring constant and L is the rest length of the spring (i.e., the length "preferred" by the edge, ignoring the repulsive force)

Force-Direct Layout of Graph

- Implementation
  - To implement this force-directed layout, assume that the nodes are stored in an array `nodes[]`, where each element of the array contains a position `x, y` and the net force `force_x, force_y` acting on the node.
  - The forces are simulated in a loop that computes the net forces at each time step and updates the positions of the nodes, hopefully until the layout converges to some good distributed positions.

Force-Direct Layout of Graph

```
L = ... // spring rest length
K_r = ... // repulsive force constant
K_s = ... // spring constant
delta_t = ... // time step

N = nodes.length

// initialize net forces
for i = 0 to N-1
   nodes[i].force_x = 0
   nodes[i].force_y = 0

// repulsion between all pairs
for i1 = 0 to N-2
   node1 = nodes[i1]
   for i2 = i1+1 to N-1
      node2 = nodes[i2]
      dx = node2.x - node1.x
      dy = node2.y - node1.y
      if dx != 0 or dy != 0
         distanceSquared = dx *dx + dy*dy
         distance = sqrt( distanceSquared )
         force = K_r / distanceSquared
         fx = force * dx / distance
         fy = force * dy / distance
         node1.force_x = node1.force_x - fx
         node1.force_y = node1.force_y - fy
         node2.force_x = node2.force_x + fx
         node2.force_y = node2.force_y + fy

// spring force between adjacent pairs
for i1 = 0 to N-1
   node1 = nodes[i1]
   for j = 0 to node1.neighbors.length-1
      i2 = node1.neighbors[j]
      node2 = nodes[i2]
      if i1 < i2
         dx = node2.x - node1.x
         dy = node2.y - node1.y
         if dx != 0 or dy != 0
            distance = sqrt( dx*dx + dy*dy )
            force = K_s*( distance - L )
            fx = force*dx / distance
            fy = force*dy / distance
            node1.force_x = node1.force_x + fx
            node1.force_y = node1.force_y + fy
            node2.force_x = node2.force_x - fx
            node2.force_y = node2.force_y - fy

// update positions
for i = 0 to N-1
   node = nodes[i]
   dx = delta_t*node.force_x
   dy = delta_t*node.force_y
   displacementSquared = dx*dx + dy*dy
   if ( displacementSquared > MAX_DISPLACEMENT_SQUARED )
      s = sqrt( MAX_DISPLACEMENT_SQUARED / displacementSquared )
      dx = dx *s
      dy = dy*s
   node.x = node.x + dx
   node.y = node.y + dy
```

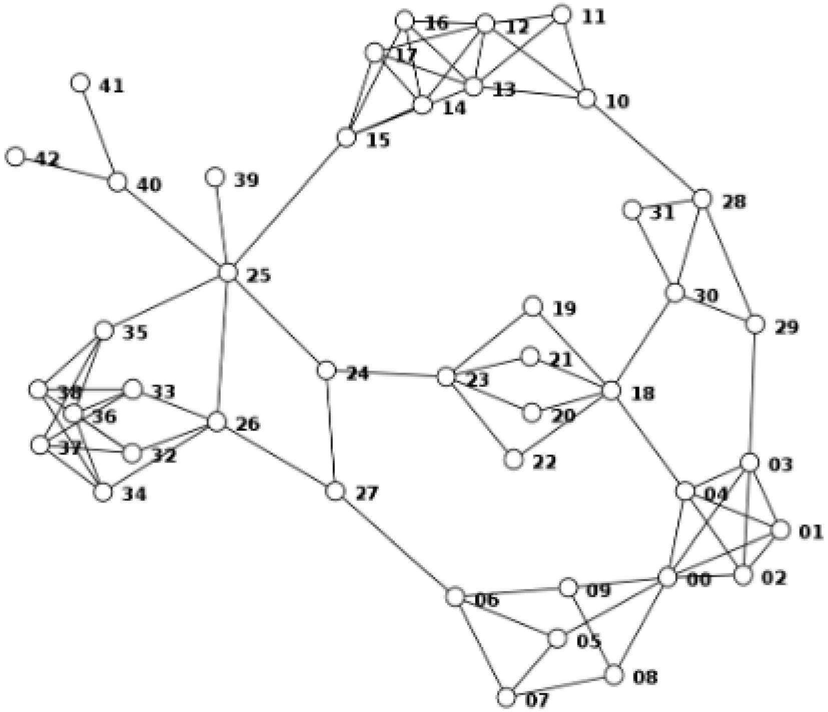
Force-Direct Layout of Graph

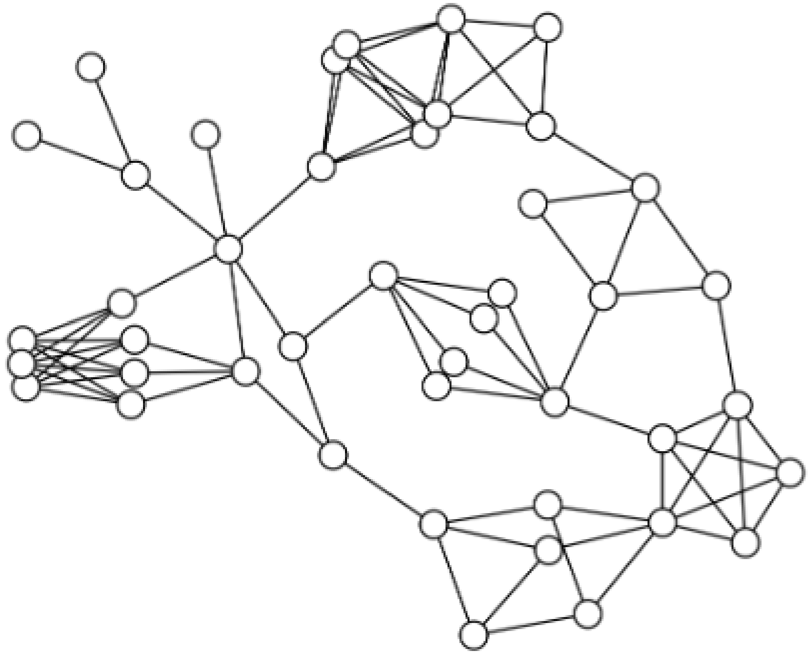
```
L = ... // spring rest length
K_r = ... // repulsive force constant
K_s = ... // spring constant
delta_t = ... // time step

N = nodes.length

// initialize net forces
for i = 0 to N-1
   nodes[i].force_x = 0
   nodes[i].force_y = 0

// repulsion between all pairs
for i1 = 0 to N-2
   node1 = nodes[i1]
   for i2 = i1+1 to N-1
      node2 = nodes[i2]
      dx = node2.x - node1.x
      dy = node2.y - node1.y
      if dx != 0 or dy != 0
         distanceSquared = dx *dx + dy*dy
         distance = sqrt( distanceSquared )
         force = K_r / distanceSquared
         fx = force * dx / distance
         fy = force * dy / distance
         node1.force_x = node1.force_x - fx
         node1.force_y = node1.force_y - fy
         node2.force_x = node2.force_x + fx
         node2.force_y = node2.force_y + fy

// spring force between adjacent pairs
for i1 = 0 to N-1
   node1 = nodes[i1]
   for j = 0 to node1.neighbors.length-1
      i2 = node1.neighbors[j]
      node2 = nodes[i2]
      if i1 < i2
         dx = node2.x - node1.x
         dy = node2.y - node1.y
         if dx != 0 or dy != 0
            distance = sqrt( dx*dx + dy*dy )
            force = K_s*( distance - L )
            fx = force*dx / distance
            fy = force*dy / distance
            node1.force_x = node1.force_x + fx
            node1.force_y = node1.force_y + fy
            node2.force_x = node2.force_x - fx
            node2.force_y = node2.force_y - fy

// update positions
for i = 0 to N-1
   node = nodes[i]
   dx = delta_t*node.force_x
   dy = delta_t*node.force_y
   displacementSquared = dx*dx + dy*dy
   if ( displacementSquared > MAX_DISPLACEMENT_SQUARED )
      s = sqrt( MAX_DISPLACEMENT_SQUARED / displacementSquared )
      dx = dx *s
      dy = dy*s
   node.x = node.x + dx
   node.y = node.y + dy
```

Force-Directed Layout

- Nodes are modeled as physical bodies that are connected through springs (edges)
  - Pseudo code
  - Example
- High running time
  - The typical **force-directed** algorithms are in general *considered* to have a running time equivalent to $O(n^3)$, where n is the number of nodes of the input graph.

Force-Direct Layout of Graph





Force-directed node-link diagrams of a 43-node, 80-edge network.
Left: a low spring constant makes the edges more flexible.
Right: a high spring constant makes them more stiff

Vertex Issues

- Shape
- Color
- Size
- Location
- Label

Edge Issues

- Color
- Size
- Label
- Form
- Polyline, straight line, orthogonal, grid, curved, planar, upward/downward, ...

Force-Direct Layout of Graph

- Limitations and Improvements
  - Difficult to choose a proper *delta_t* :If the time step *delta_t* (used at lines 53, 54) is too small, many iterations will be needed to converge. On the other hand, if the time step is too large, or if the net forces generated are too large, the positions of nodes may oscillate and never converge. Line 56 imposes a limit on such movement.
  - As a minor optimization, line 56 compares squares (i.e., displacementSquared>MAX_DISPLACEMENT_SQUARED rather than displacement >MAX_DISPLACEMENT), to avoid the cost of computing a square root (unless the if succeeds)

Force-Direct Layout of Graph

- Limitations and Improvements
  - The GEM[16] algorithm speeds up convergence by decreasing a “temperature” parameter as the layout progresses, allowing nodes to <u>move larger distances earlier in the process</u>, and then constraining their movements progressively toward the end.

Force-Direct Layout of Graph

- Limitations and Improvements
  - A minor improvement to the above pseudocode would be to detect if the distance between two nodes is zero (by adding an else clause to the if statement at line 20), and in that case to generate a small force between the two nodes in some random direction, to push them apart. Without this, if the two nodes happen to have the same neighbors, they may remain forever “stuck” to each other.

Force-Direct Layout of Graph

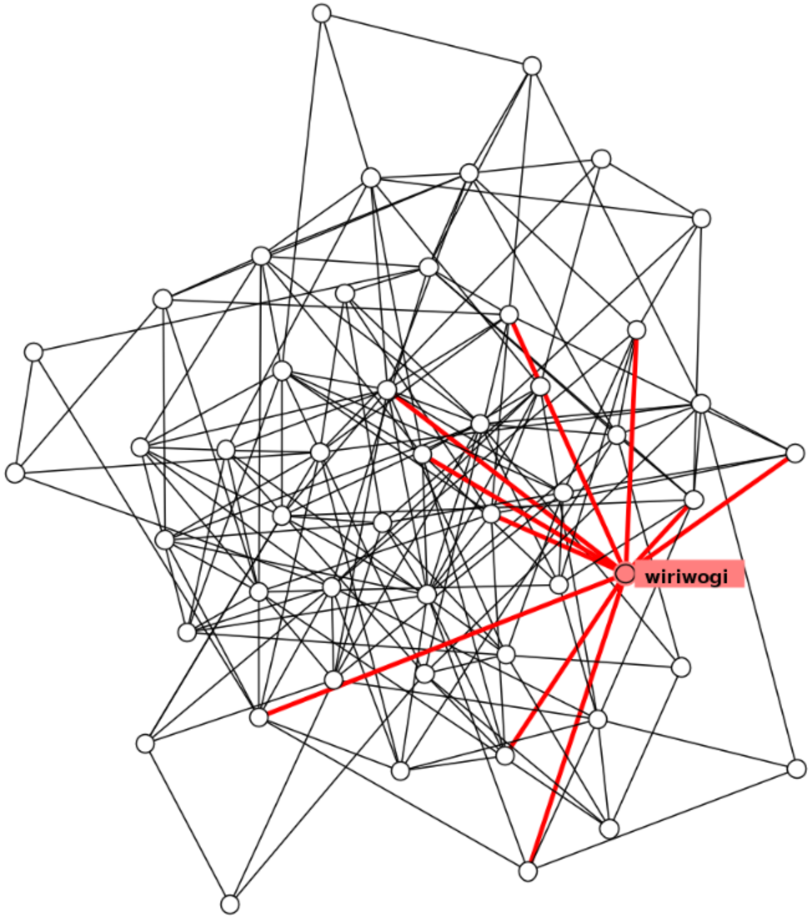
- Limitations and Improvements
  - There are infinitely many pairs of $(Kr\ ,Ks)$ values that cause the layout to converge to the same final “shape” (i.e., the same angles between edges, differing only in edge lengths). A simpler user interface would allow the user to change a single parameter corresponding to a kind of ratio of the strength of the two forces. The final shape of the layout will depend on both Kr/Ks and L.

Force-Direct Layout of Graph

- In the pseudocode above, the computation of repulsive forces is a bottleneck, since *it requires $O(N^2)$* time, where N is the number of nodes.

- Possible solution:
  - We could eliminate the repulsive force, and instead simulate springs of length L between all adjacent nodes, as well as springs of length $2L$ between all nodes that are two edges apart, and possibly springs of length $3L$ between nodes that are three edges apart, etc., up to some limit. The extra springs would help to spread apart the network, as did the original repulsive forces. As long as the number of edges is not too high, and there aren't too many springs, the computation time may be much less than $O(N^2)$.

Force-Direct Layout of Graph



Force-directed node-link diagram of a random 50-node, 200-edge graph.

Limitations and Improvements

As can be seen in the left example, the multiple crossings of edges can <u>make it unclear when certain edges pass close to a node or are connected to a node</u>. Also, in such layouts where the nodes are rather closely packed, there isn't much room left to display labels or other information associated with each node

This leads to the next layout method

What’s the Problem?

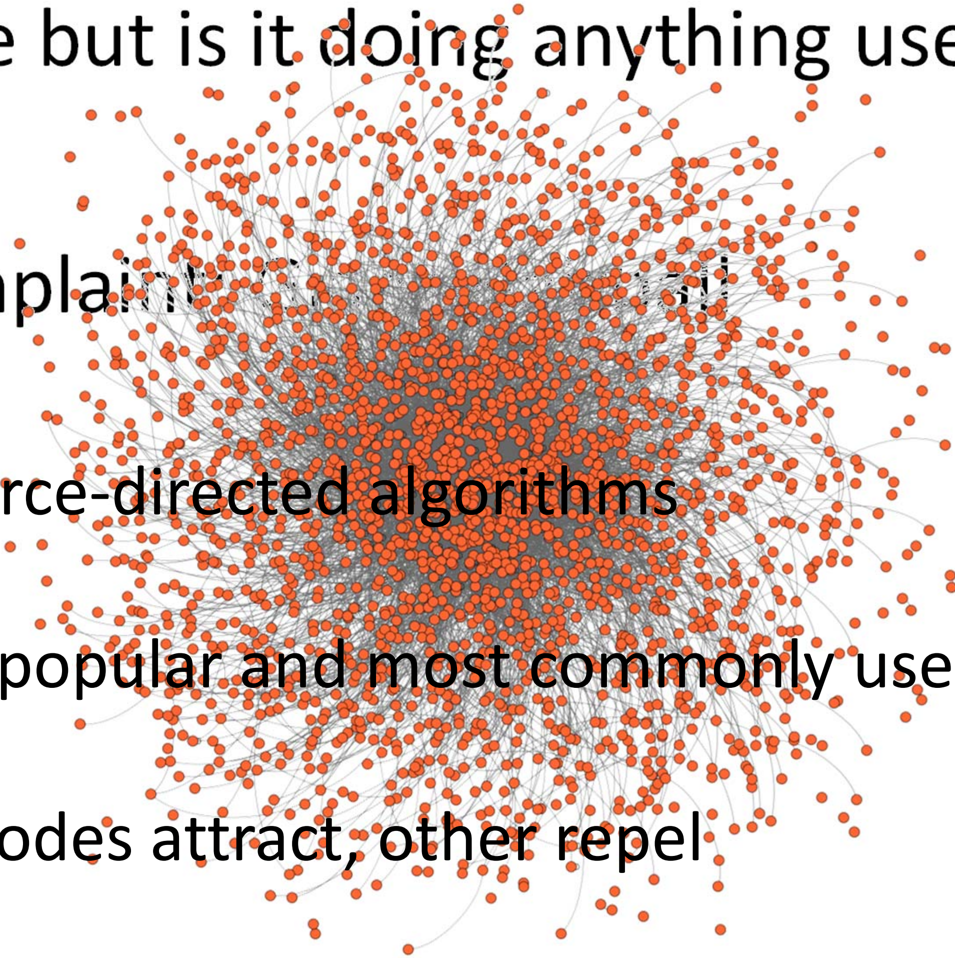
It looks nice but is it doing anything useful?

Typical complaint:

Caused by force-directed algorithms

Old, but still popular and most commonly used

Connected nodes attract, other repel



Yeast interaction network in Gephi

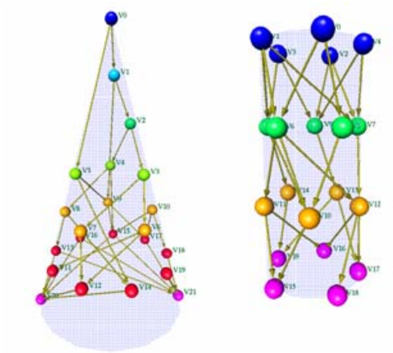
The problem of visualizing large graphs

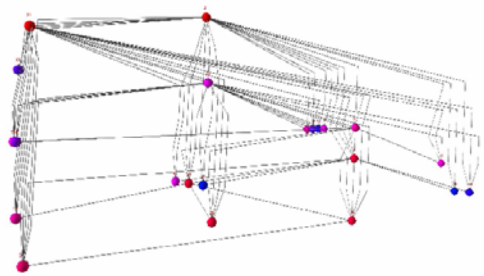
Some major issues in the visualization of large graphs:

- Readability: optimization of aesthetic criteria
- Scalability: fast computation
- Visual complexity: interaction tools that allow users to limit the amount of information displayed on the screen
  - overview of the graph
  - details on demand
  - user’s mental map preservation

Brainstorming Exercise 1

- How could we scale graph layouts to more than a few hundred nodes?
  - Possible strategy: use 3D
    - Why or why not?





Basic Graph Layout Techniques

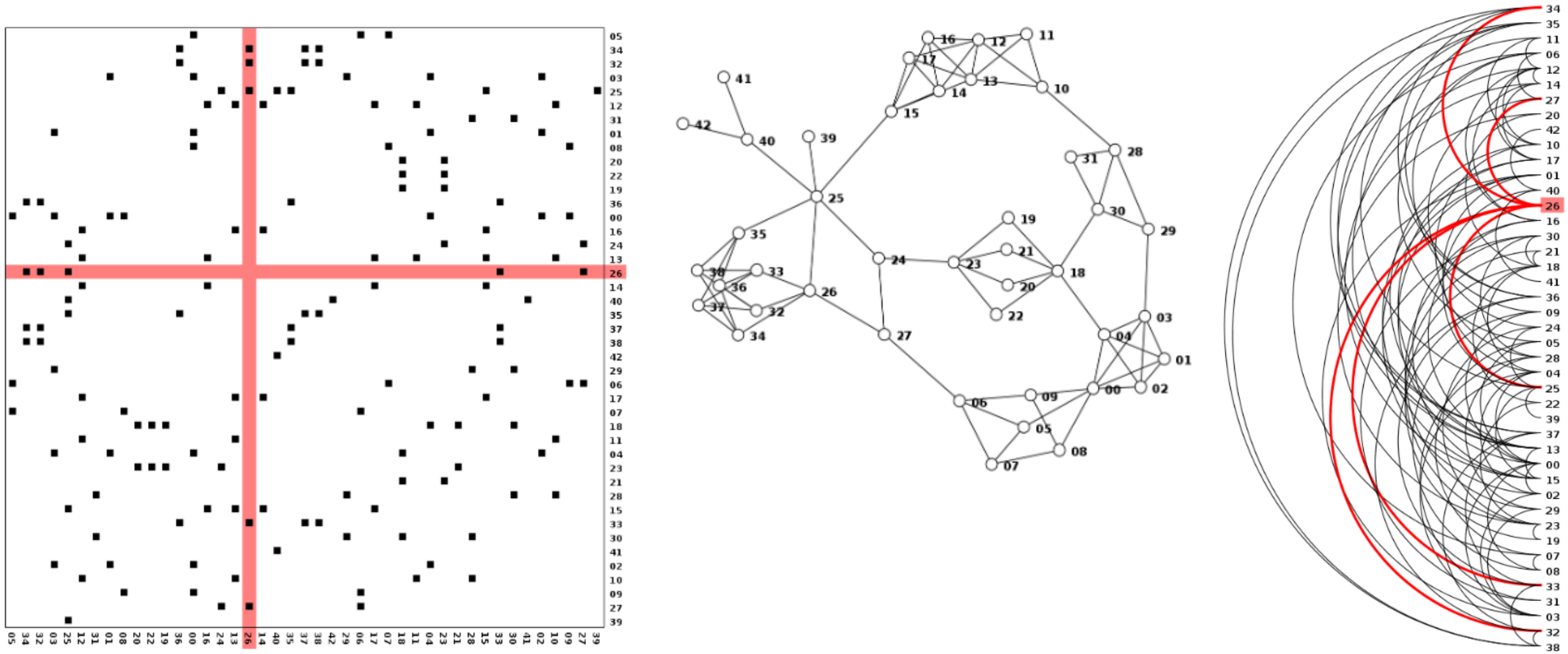
- Force-direct layout
- Adjacent matrix
- Arc-diagram
- Circular layout

Adjacent Matrix Representations

- An adjacent matrix contains one row and one column for each node of a network.
  - Given two nodes i and j, the entry located at (i, j) and (j, i) in the matrix contain information about the edge(s) between the two nodes.
  - Typically, each cell contains a boolean value indicating if an edge exists between the two nodes.
  - If the graph is undirected, the matrix is symmetric.

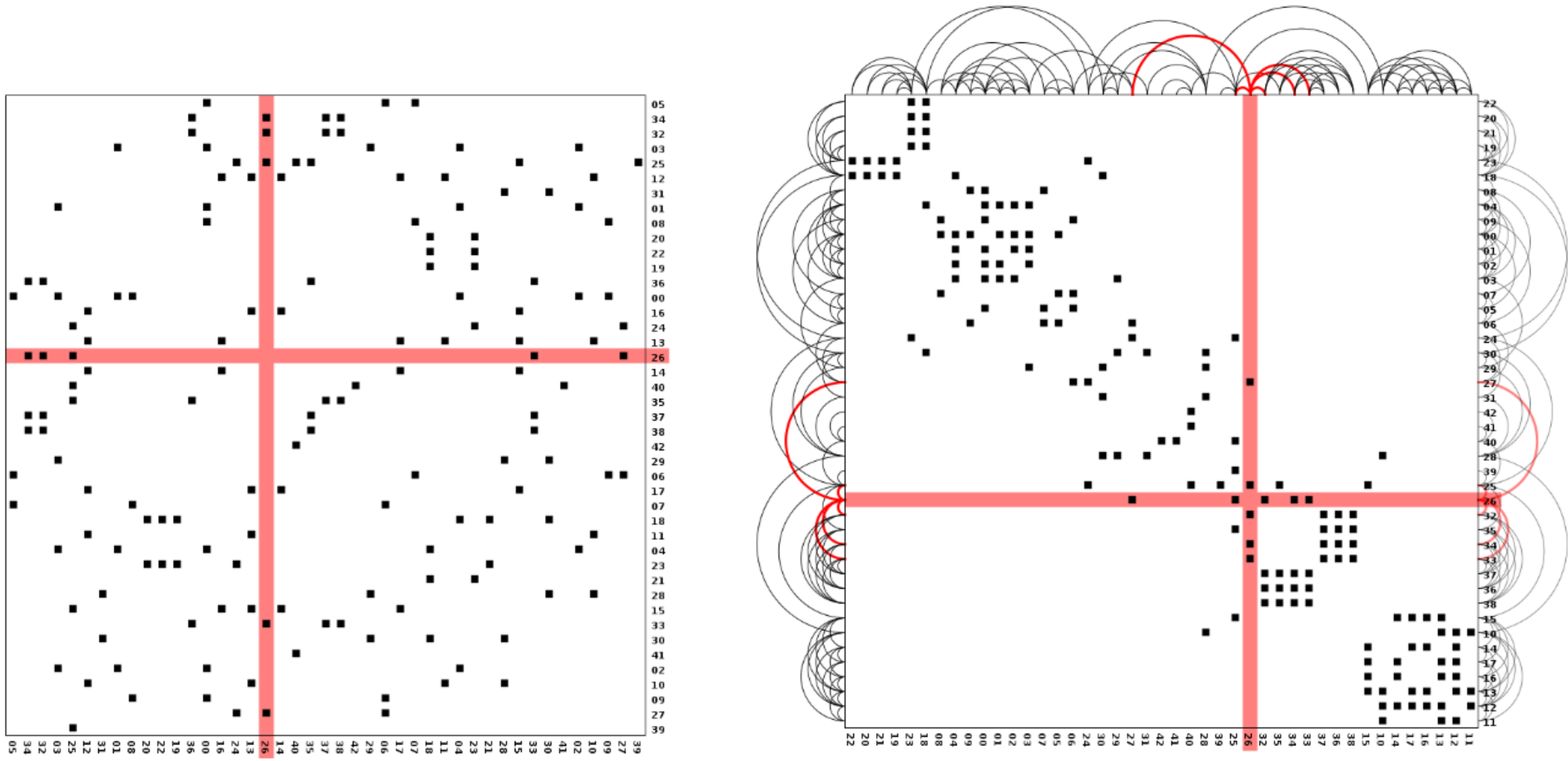
- Pros:
  - Visualizing a network as a matrix has the advantage of eliminating all edge crossings, since the edges correspond to non-overlapping entries.
- Cons:
  - The ordering of rows and columns greatly influences how easy it is to interpret the matrix.
  - Difficult to follow a path in the graph.
  - Limited by screen resolution.

An Example



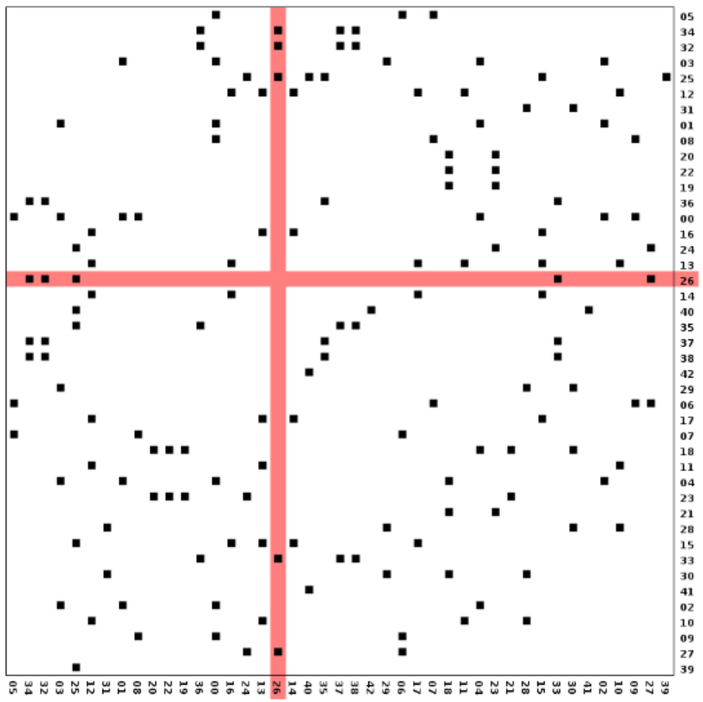
Adjacency matrix visualizations of a 43-node, 80-edge network. Left: with a random ordering of rows and columns.

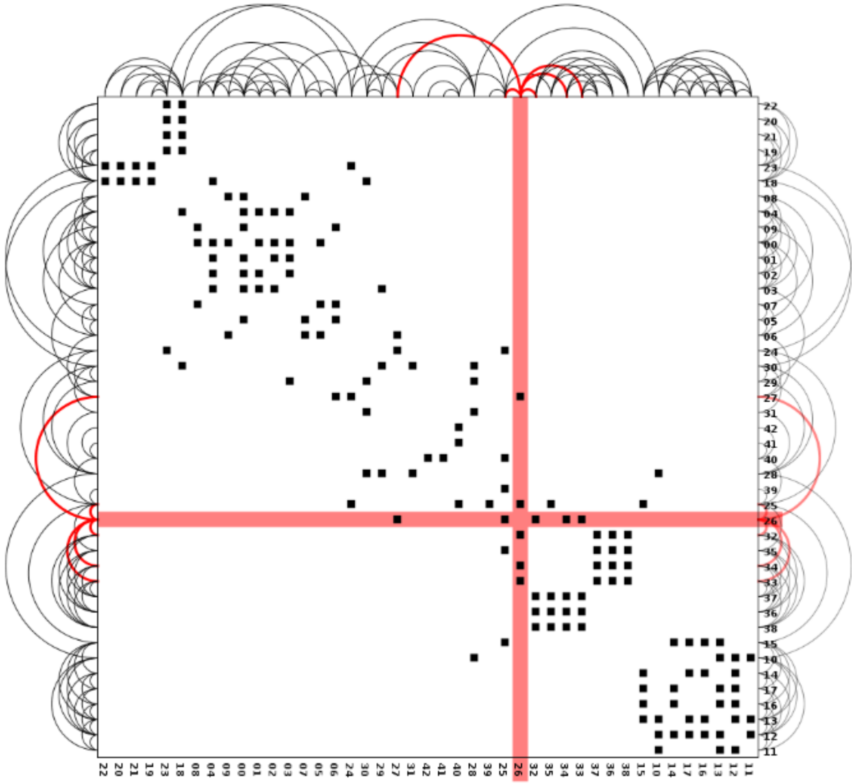
An Example



Adjacency matrix visualizations of a 43-node, 80-edge network. Left: with a random ordering of rows and columns. Right: after **barycenter ordering** and adding arc diagrams. The multiple arc diagrams are redundant, but reduce the distance of eye movements from the inside of the matrix to the nearest arcs.

An Example

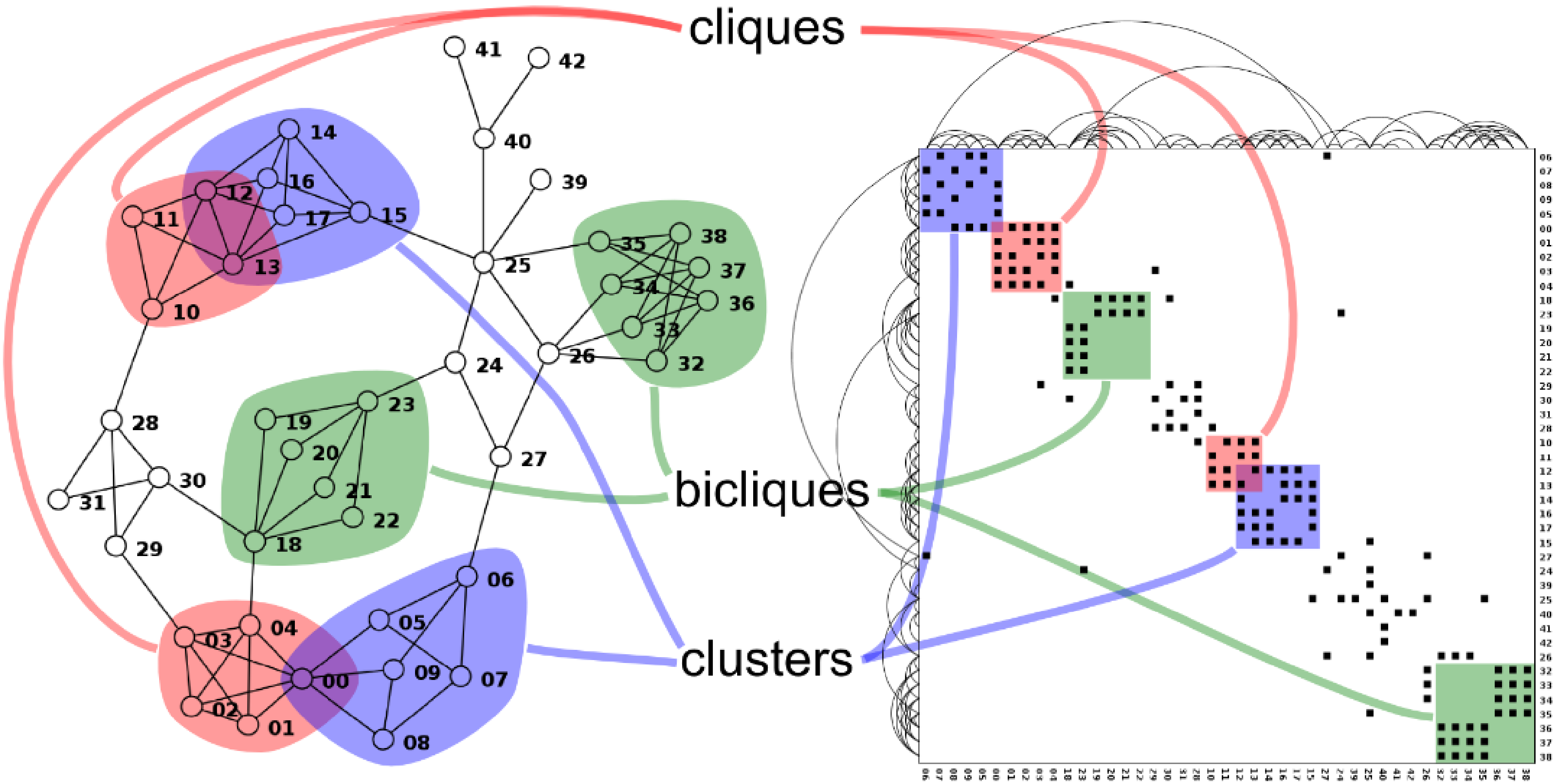




Interestingly, by bringing nodes “closer” to their neighbors with the **barycenter heuristic**, this pushes the edges (filled-in matrix cells) **closer to the diagonal of the matrix**, making certain **patterns** appear in the positions of the cells.

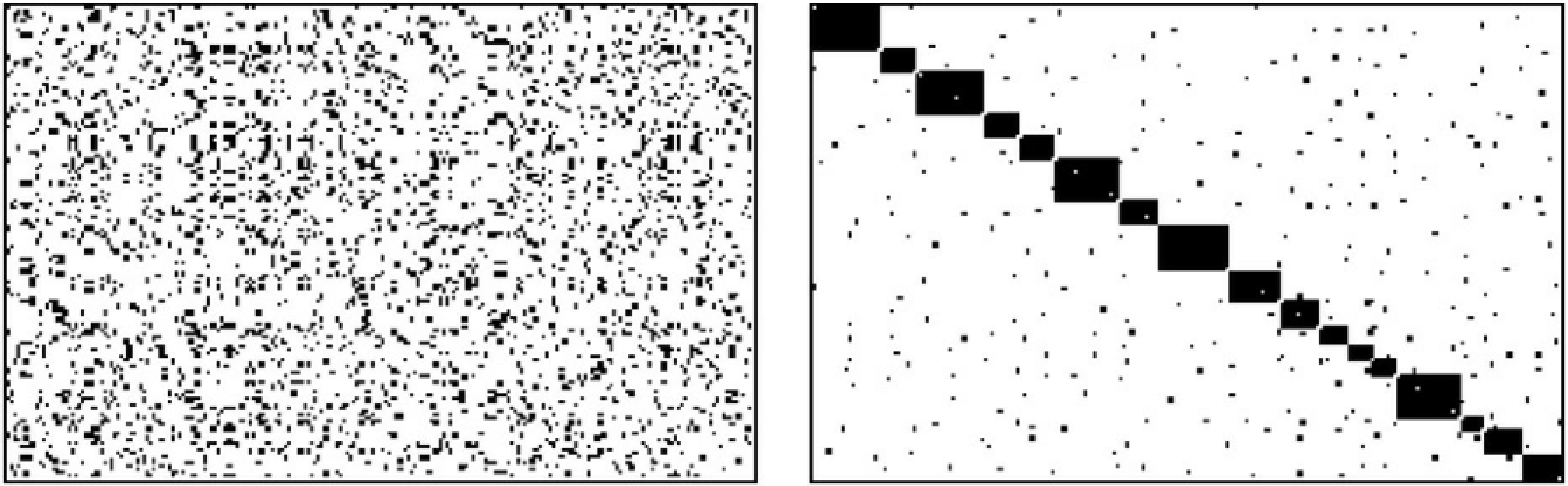
Adjacent Matrix Representations

Certain **subgraphs** (subsets of nodes and edges in the graph) correspond to easy-to-recognize patterns in the adjacency matrix, given an appropriate ordering of rows and columns.



Patterns corresponding to interesting subgraphs appear **along the diagonal** of an appropriately ordered adjacency matrix (say, via barycenter ordering)

Another Example



The adjacency matrix of a 210-vertex graph with 1505 edges composed of 17 dense clusters. On the left, the vertices are ordered randomly and the graph structure can hardly be observed. On the right, the vertex ordering is by cluster and the 17-cluster structure is evident. Each black dot corresponds to an element of the adjacency matrix that has the value one, the white areas correspond to elements with the value zero.

<u>Matrix diagonalization in itself is an important application of clustering algorithms.</u>

Other advantages:

Matrices have the added advantage of also being able to display information related to each edge within the entries of the matrix. For example, if the edges are weighted, this weight can be shown in the color of the entry.

Entries can also contain small graphics or glyphs, as in Brandes and Nick's "gestaltmatrix" where each entry contains a glyph showing the evolution of the edge over time.

Limitations:

An important disadvantage of using adjacency matrices, however, is that the **space they require is $\boldsymbol{O}(\boldsymbol{N}^2)$** where N is the number of nodes.

Brainstorming Exercise 2

- Which graph analysis tasks are better supported by the node-link view, and which are better supported by the matrix view?
- How does the above answer change with increasing size of the graph?

Graph Visualization Tools

- Sigma.js JavaScript library
- Gephi open source graph viz platform
- Many more!

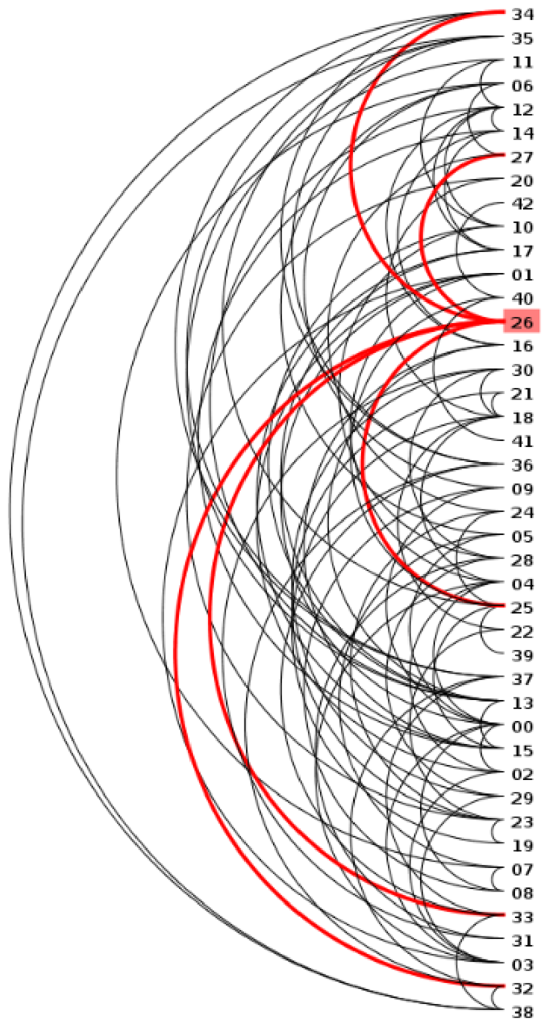
Basic Graph Layout Techniques

- Force-direct layout
- Adjacent matrix
- Arc-diagram
- Circular layout

Arc Diagrams and Barycenter Ordering

- It is sometimes useful to layout the nodes of a network along a straight line, in what might be called **linearization**. With such a layout, edges can be drawn as circular arcs, yielding an *arc diagram*.

- It is important that the arcs in the diagram all cover the same angle, such as 180 degrees. This way, an arc between nodes n1 and n2 will extend outward by a distance proportional to the distance between n1 and n2, making it easier to disambiguate the arcs.

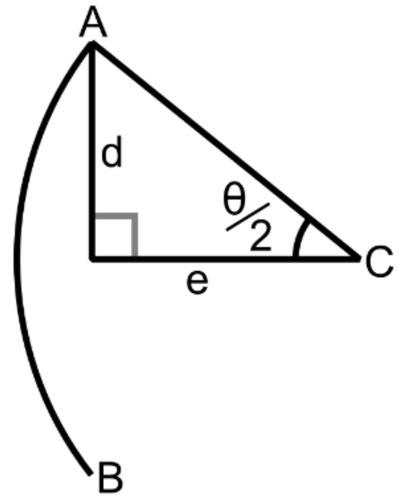


Arc diagrams of a 43-node, 80-edge network

Arc Diagrams and Barycenter Ordering

To program a subroutine that draws an arc covering angle θ connecting points $A = (x, y_1)$ and $B = (x, y_2)$, we need to find the center C of the arc.

Image to the right shows a right triangle connecting A, C and the midpoint between A and B. The length of one side of the triangle is $d = |y_1 - y_2|/2$, and we also have $\tan\left(\frac{\theta}{2}\right) = d/e$, hence $C = (x + e, \frac{y_1+y_2}{2})$ where $e = d/(\tan\left(\frac{\theta}{2}\right))$.

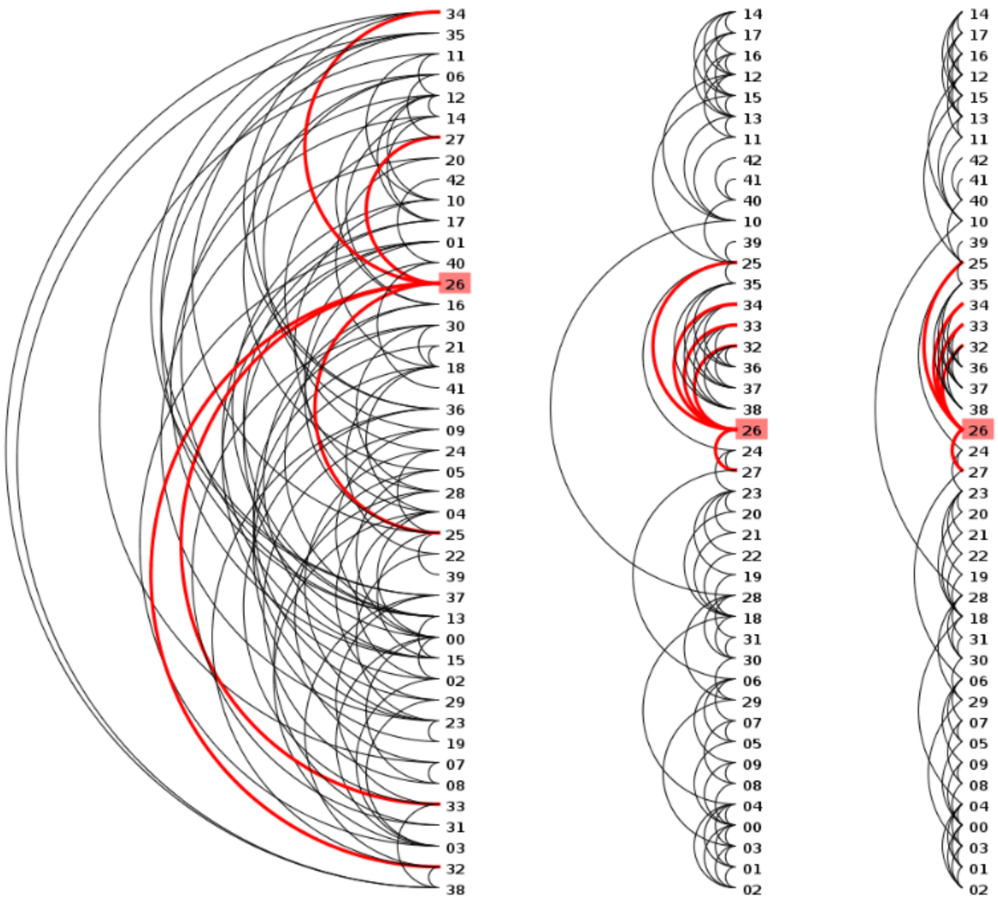


An arc covering angle θ, with center C

Arc Diagrams and Barycenter Ordering

Sorting the nodes:

We might order the nodes to reduce the total length of the arcs, making the *topology* of the network easier to understand.



Left: with a random ordering and 180-degree arcs.
Middle: after applying the barycenter heuristic to order the nodes.
Right: after changing the angles of the arcs to 100 degrees.

Arc Diagrams and Barycenter Ordering

Sorting the nodes:

We might order the nodes to reduce the total length of the arcs, making the *topology* of the network easier to understand. There are many algorithms for computing such an ordering. However, we will discuss an easy-to-program technique called the barycenter heuristic.

The barycenter heuristic is an iterative technique where we compute the *average position* (or “barycenter”) of the neighbors of each node, and then sort the nodes by this average position, and then repeat. Intuitively, this should move nodes closer to their neighbors, making the arcs shorter.

Arc Diagrams and Barycenter Ordering

An implementation of barycenter heuristic method:

we will assume that the `nodes[]` array is fixed, and use a second data structure, called `orderedNodes[]`, to store the current ordering of nodes to use for the arc diagram.

We will use the term *index* to refer to a node's fixed location within `nodes[]`, and position to refer to the node's current location within `orderedNodes[]`. Each element of `orderedNodes[]` will store an *index* and an average. For example, if `orderedNodes[3].index == 7`, then `orderedNodes[3]` corresponds to `nodes[7]`, and `nodes[7]` is to be displayed at position 3 in the arc diagram. To find the index corresponding to a given position, we can simply perform a look-up in `orderedNodes[]`. To perform an inverse look-up, we define a function that computes the position p of a node given its index `i`:

```
function positionOfNode( i )
   for p = 0 to N-1
      if orderedNodes[p].index == i
         return p
```

Arc Diagrams and Barycenter Ordering

Given the `positionOfNode()`, we can implement the inner body of the barycenter heuristic like the following:

```
1 // compute average position of neighbors
2 for i1 = 0 to N-1
3    node1 = nodes[i1]
4    p1 = positionOfNode(i1)
5    sum = p1
6    for j = 0 to node1.neighbors.length-1
7       i2 = node1.neighbors[j]
8       node2 = nodes[i2]
9       p2 = positionOfNode(i2)
10      sum = sum + p2
11   orderedNodes[p1].average = sum/ (node1.neighbors.length + 1)
12
13 // sort the array according to the values of average
14 sort( orderedNodes, comparator )
```

```
function positionOfNode( i )
   for p = 0 to N-1
      if orderedNodes[p].index == i
         return p
```

Lines 1 through 14 would be inside a loop that iterates several times, hopefully until convergence to a near-optimal ordering.

Arc Diagrams and Barycenter Ordering

In practice, rather than converging, the algorithm sometimes enters a cycle. Thus, a limit on the number of iterations should be imposed, stopping the loop if the limit is reached (one rule of thumb is to limit the number of iterations to kN, where N is the number of nodes and k is a small positive constant). Simple ways to improve the algorithm would be to (1) detect if it has converged to an ordering that does not change with additional iterations, and in such a case stop the loop; (2) detect cycles, and similarly stop the loop.

Line 14 of the pseudo-code sorts the contents of `orderedNodes[]` according to a comparator defined by the calling code. Typical programming environments provide an efficient $O\ (NlogN)$ implementation of sort (such as *qsort* in C).

Arc Diagrams and Barycenter Ordering

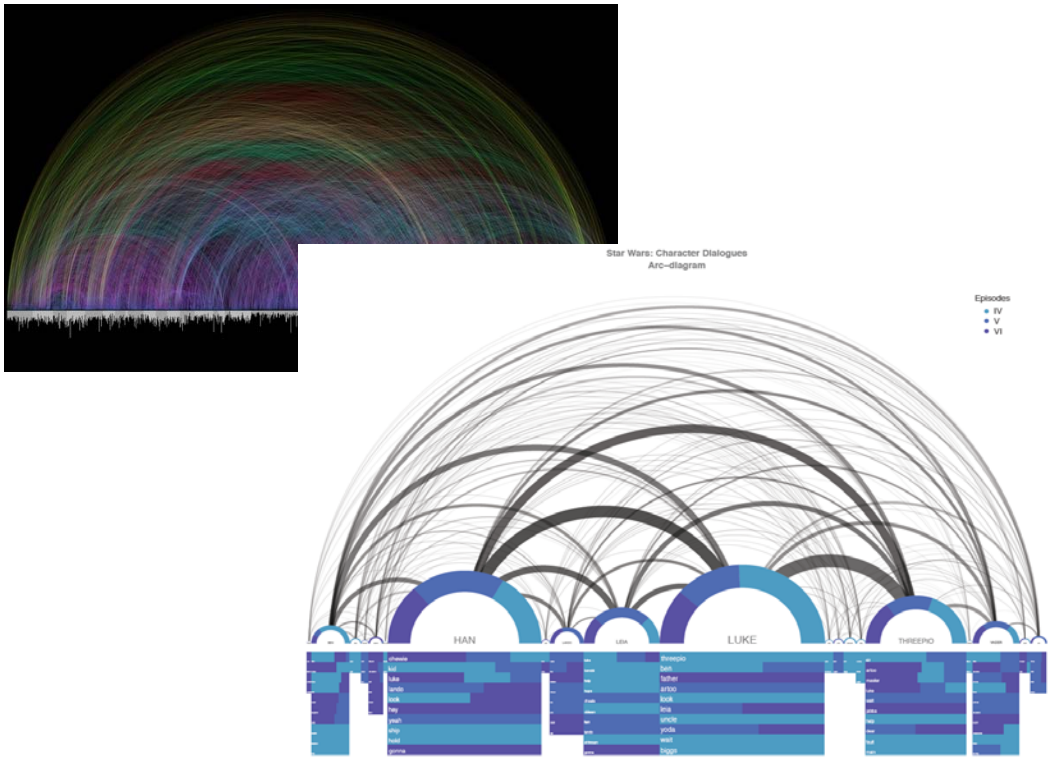
Other sorting of the nodes:

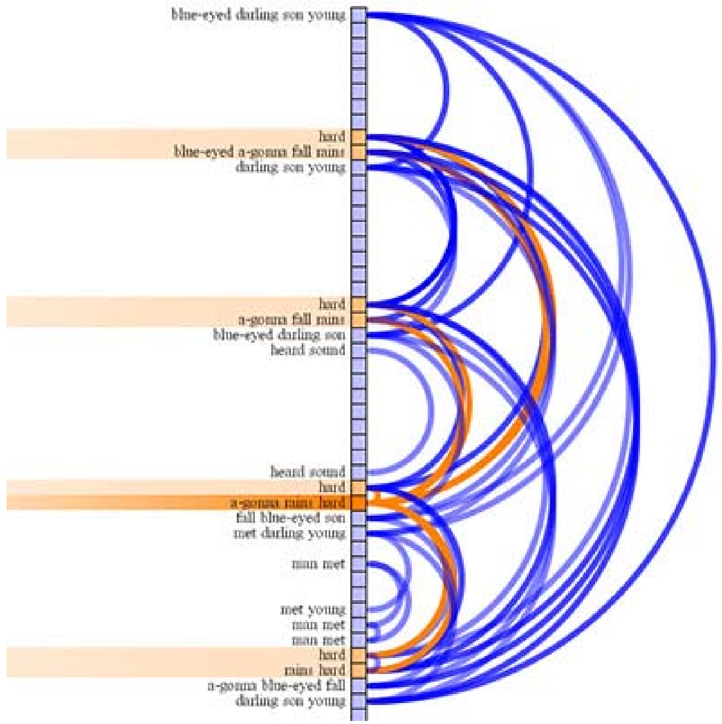
The nodes within an arc diagram might be sorted in other ways. For example, if each node has an associated label, and represents an object with a size, time-stamp, or other attribute, the nodes in the arc diagram might be sorted alphabetically, or by size, time, etc., helping the user to analyze the network. Furthermore, every node has a degree, as well as additional metrics that can be computed, and any of these might be used to sort the nodes within the linear ordering of an arc diagram.

Arc Diagrams and Barycenter Ordering

The linear arrangement of nodes in an arc diagram has many advantages.

As already mentioned, there is room to the right of each node for a long text label, if desired. The space to the right of nodes can also be used to display small graphics, such as line charts for each node, possibly to show a quantity associated with the node that evolves with time.



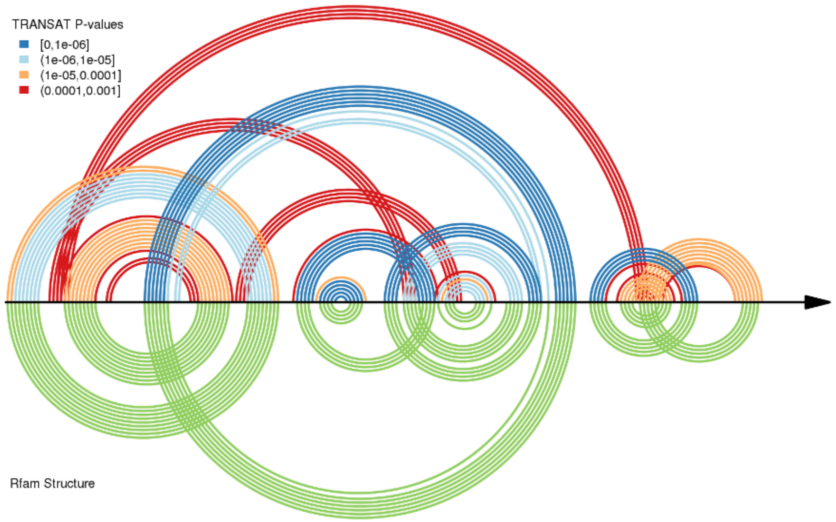


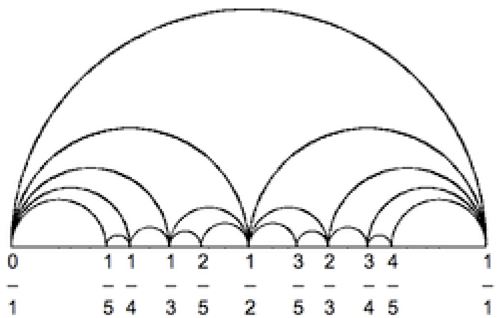
Arc Diagrams and Barycenter Ordering

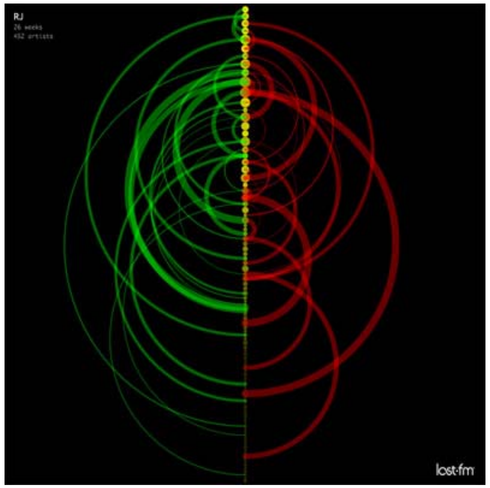
The linear arrangement of nodes in an arc diagram has many advantages.

As already mentioned, there is room to the right of each node for a long text label, if desired. The space to the right of nodes can also be used to display small graphics, such as line charts for each node, possibly to show a quantity associated with the node that evolves with time.

Arc diagrams can also be incorporated as an axis within a larger graphic or visualization







Arc Diagrams and Barycenter Ordering

The linear arrangement of nodes in an arc diagram has many advantages.

As already mentioned, there is room to the right of each node for a long text label, if desired. The space to the right of nodes can also be used to display small graphics, such as line charts for each node, possibly to show a quantity associated with the node that evolves with time.

Arc diagrams can also be incorporated as an axis within a larger graphic or visualization

Also, as mentioned, the nodes within an arc diagram can be sorted in different ways, which can be useful for seeing relationships between nodes with specific attribute values.

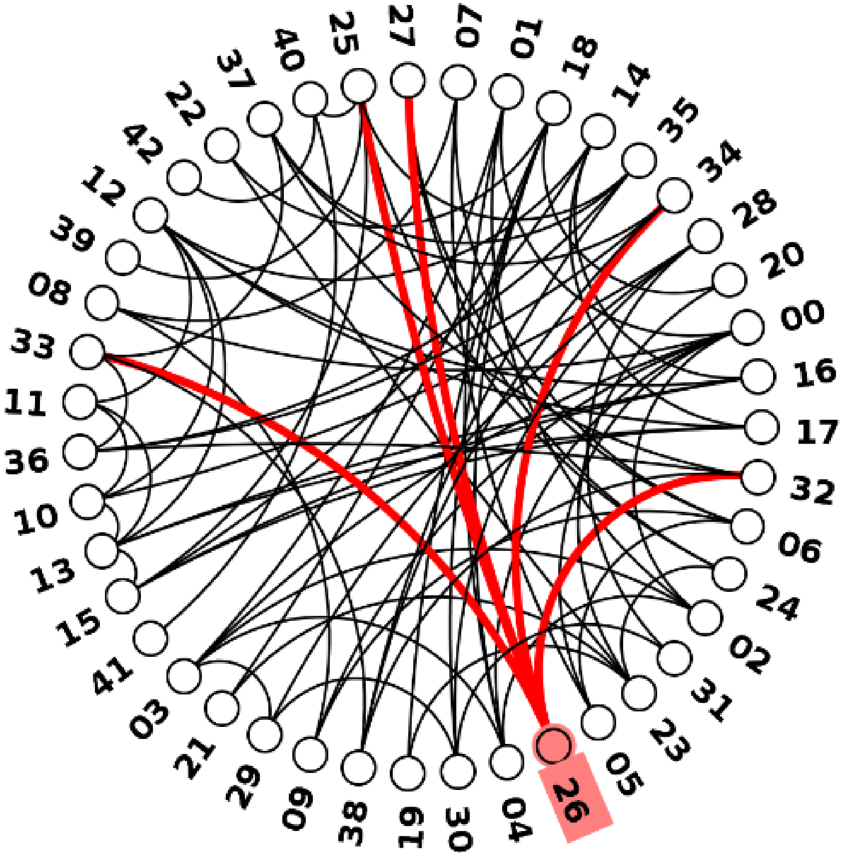
Despite the advantages of arc diagrams, and the room available to draw labels beside nodes, if there are too many edges that cross each other, it becomes difficult to read the edges. We next introduce an alternative visualization technique that eliminates edge crossings.

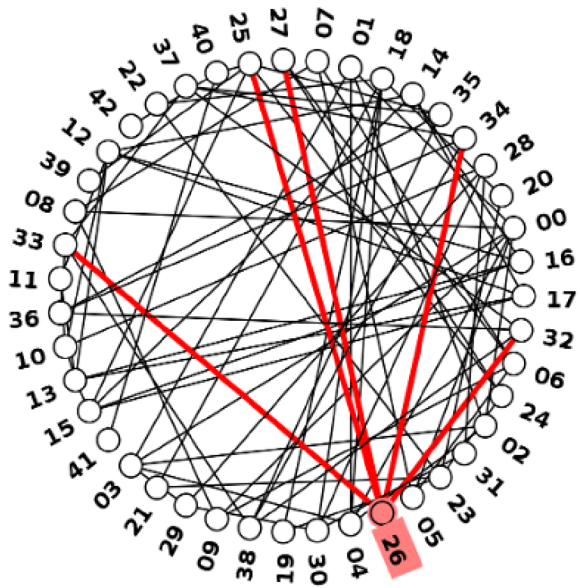
Basic Graph Layout Techniques

- Force-direct layout
- Adjacent matrix
- Arc-diagram
- Circular layout

Circular Layouts

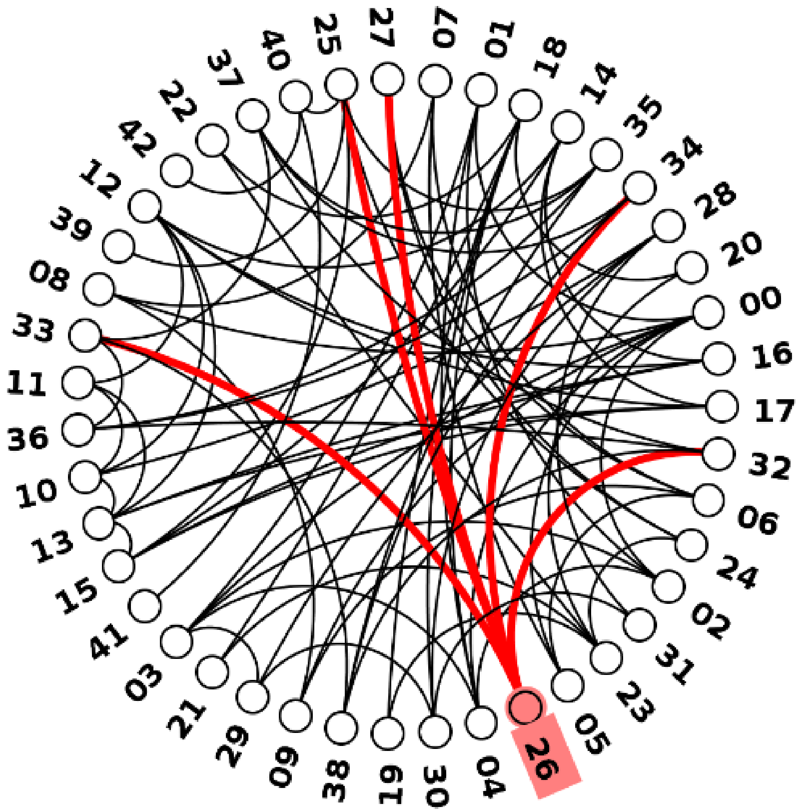
- Position nodes on the circumference of a circle, while the edges are drawn as curves rather than straight lines.

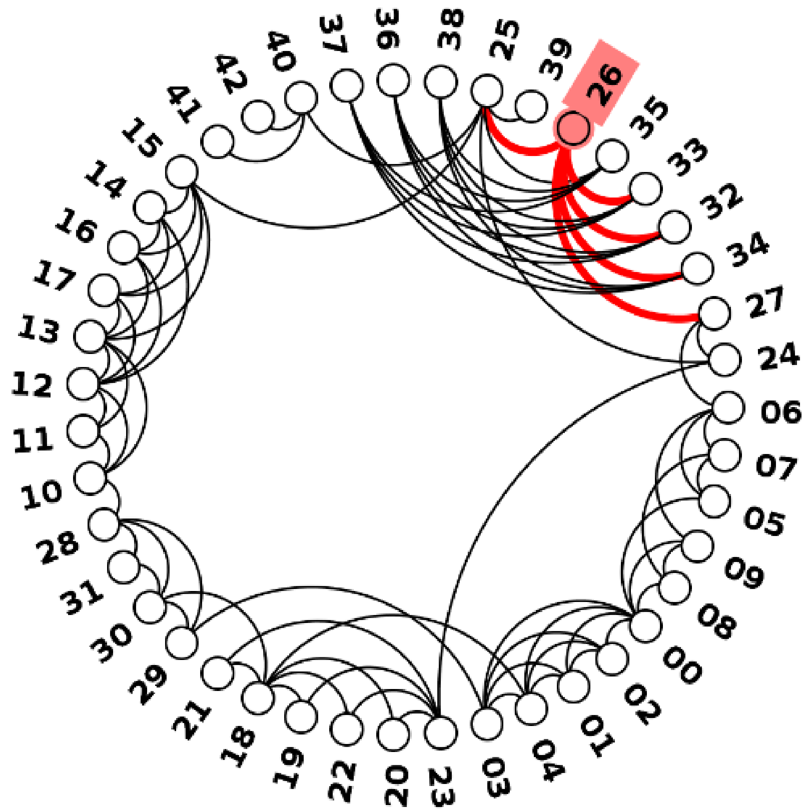




Circular Layouts

Again, the order chosen for the nodes greatly influences how clear the visualization is. The barycenter heuristic can be applied again to this layout, with a slight modification to account for the “wrap around” of the circular layout.





Circular Layouts

To correctly adapt the barycenter heuristic to this layout, consider how to compute the "average position" of the neighbors of a node.

As an example, if one neighbor is positioned at an angle of 10 degrees, and another is at an angle of 350 degrees, simply taking the numerical average yields $(10 + 350)/2 = 180$ degrees, whereas the intuitively correct barycenter is at 0 degree (or 360 degrees).

So, to correctly compute the barycenter, we do not compute averages of angles. Instead, we convert each node to a unit vector in the appropriate direction, add these unit vectors together, and find the angle of the vector sum.

Define the function `angle(p)=p*2*pi/N` giving the angle of a node at position p. Then, the pseudo-code for the barycenter heuristic becomes

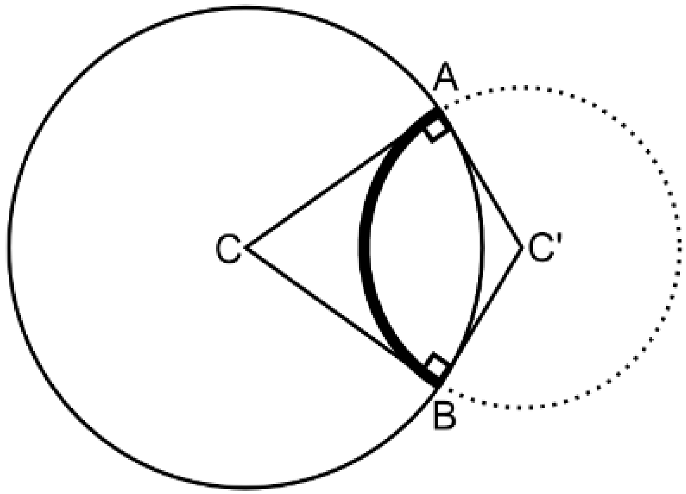
```
// compute average position of neighbors
for i1 = 0 to N-1
      node1 = nodes[i1]
      p1 = positionOfNode(i1)
      sum_x = cos(angle(p1))
      sum_y = sin(angle(p1))
      for j = 0 to node1.neighbors.length-1
           i2 = node1.neighbors[j]
           node2 = nodes[i2]
           p2 = positionOfNode(i2)
           sum_x = sum_x + cos(angle(p2))
           sum_y = sum_y + sin(angle(p2))
         orderedNodes[p1].average
               = angleOfVector(sum_x,sum_y)

// sort the array according to the values of average
sort( orderedNodes, comparator)
```

```
function angleOfVector( x, y )
    hypotenuse = sqrt( x*x + y*y )
    theta = arcsin( y / hypotenuse )
    if x < 0
       theta = pi - theta
    // Now theta is in [-pi/2,3*pi/2]
    if theta < 0
       theta = theta + 2*pi
    // Now theta is in [0,2*pi]
    return theta
```

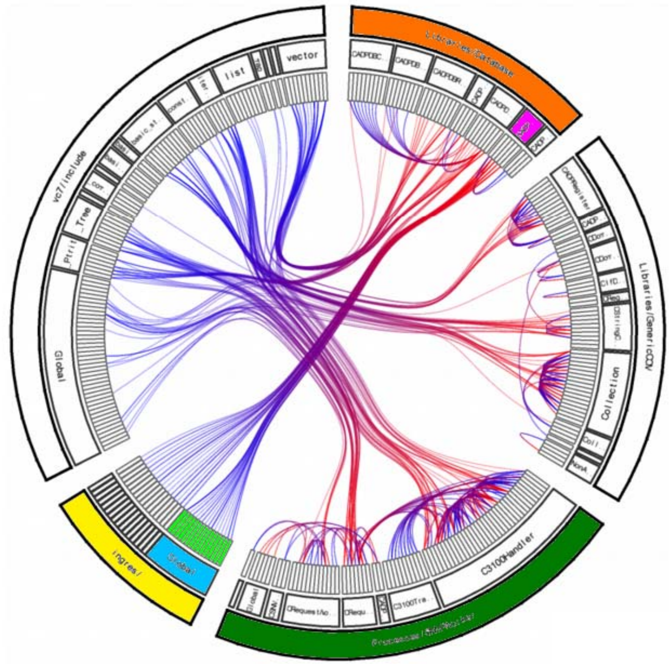
Circular Layouts

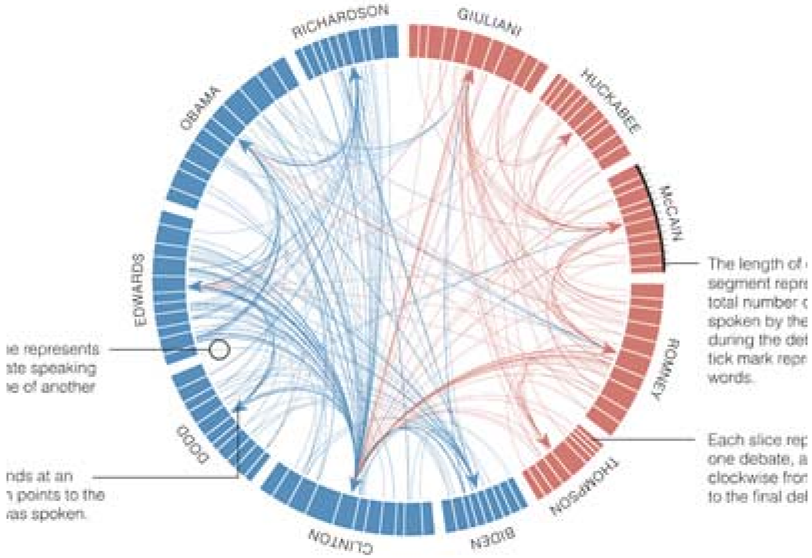
Let C be the center of the circular layout.

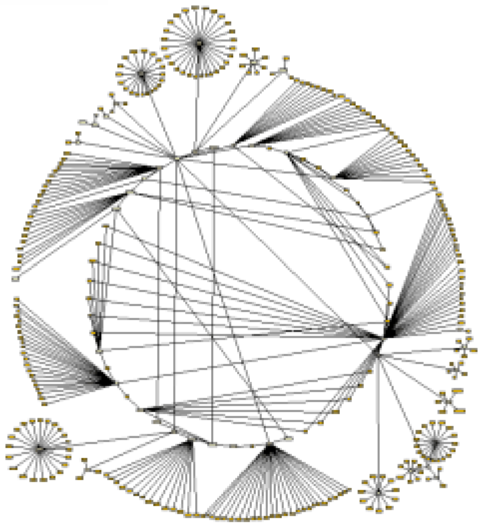
To draw a curved arc between A and B on the circumference, we draw a circular arc that is tangent to the lines AC and BC. The center C' of the arc can be found by finding the intersection between a line through A that is perpendicular to AC, and a line through B that is perpendicular to BC.

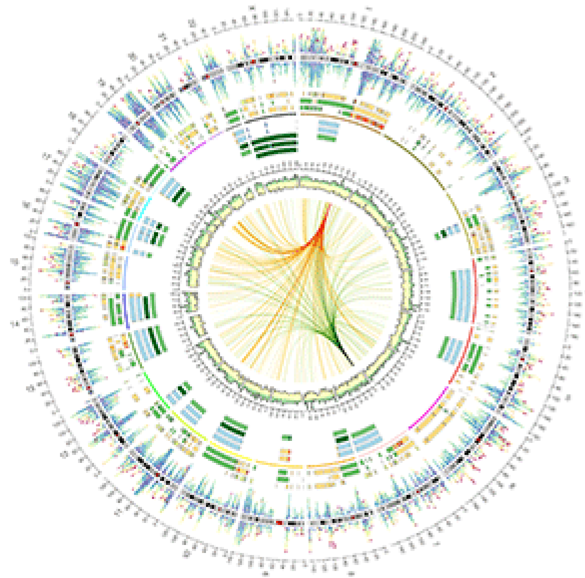


More Examples





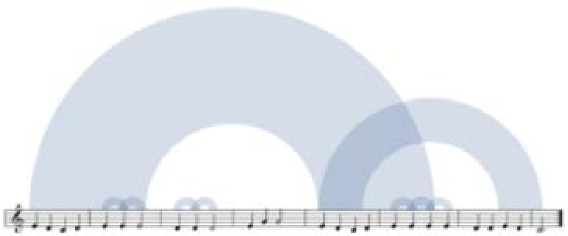


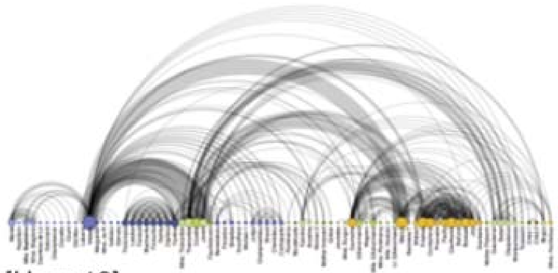


Comparison of Layout Techniques

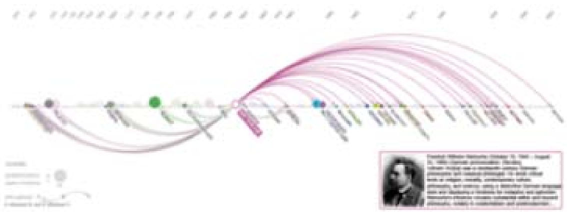
	node-link diagram	circular layout	arc diagram	adjacency matrix	MatLink
Height of each node's label	$O(1/\sqrt{N})$ (best)	$O(\pi/N)$	$O(1/N)$	$O(k_1/N)$	$O(k_2/N)$ (worst)
Easy to perceive paths	yes	somewhat	somewhat	no	somewhat
Avoids edge crossings	no	no	no	yes	yes
Avoids ambiguity from edges passing close to nodes	no	yes	yes	yes	yes
Can depict an ordering of nodes	no	yes	yes	yes	yes
Can depict information about each edge	somewhat	somewhat	somewhat	yes	yes
Node labels all have the same orientation, for easier reading	yes	no	yes	yes	yes

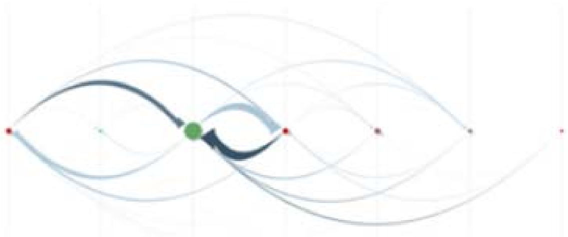
Arc diagrams

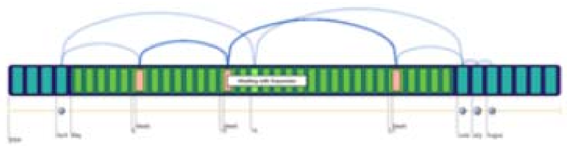

[Wattenberg 02]

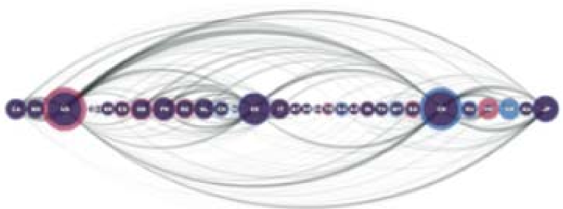

[Heer 10]


[Nagel 12]

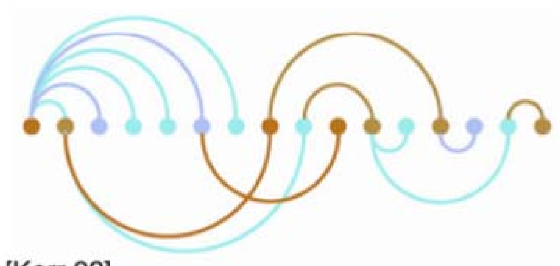

[Dörk 12]


[Wattenberg 06]


[Neumann 05]


[Stefaner 12]


[Steinweber 07]


[Kerr 03]