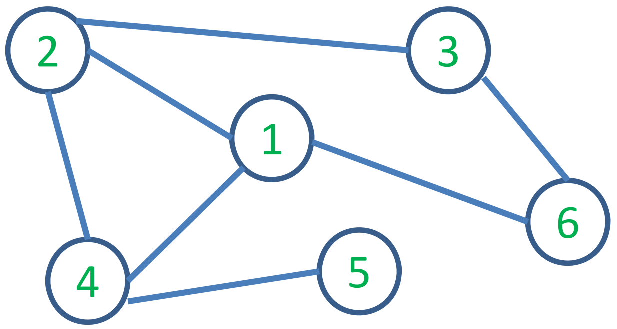
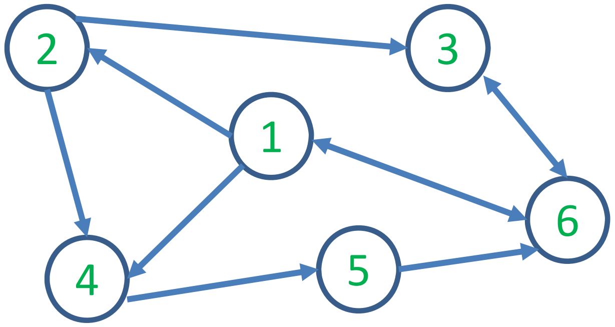
Graph Visualization

Goals: understand important concepts and characteristics of graphs; know the classic graph layout and visualization techniques

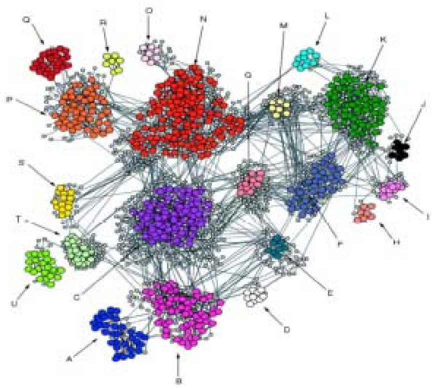
What is a Graph

- 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.



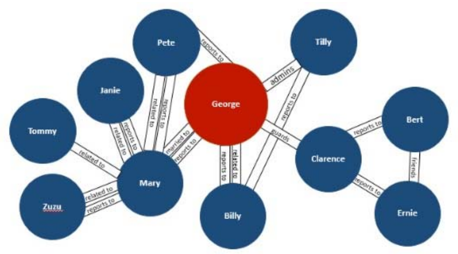
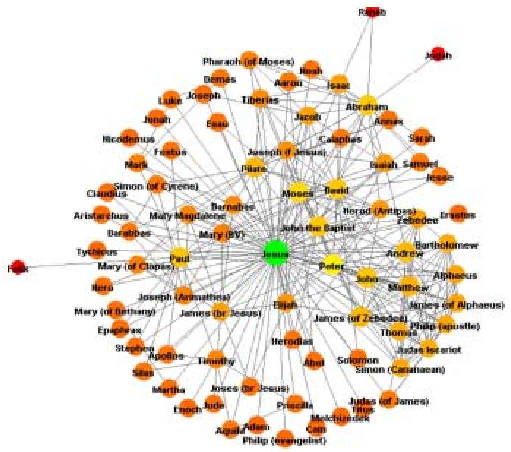


Graphs are everywhere

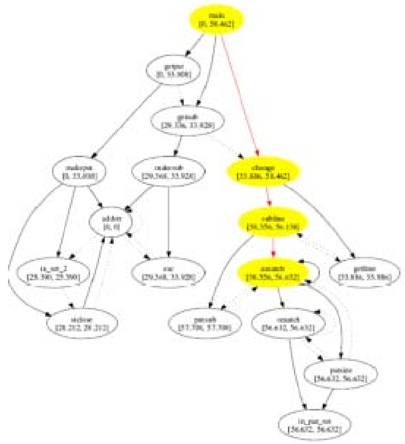


Magwene *et al. Genome Biology* 2004 **5**:R100

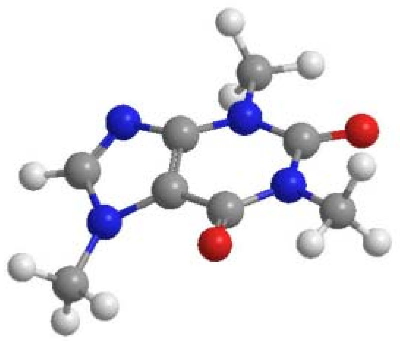
Co-expression Network

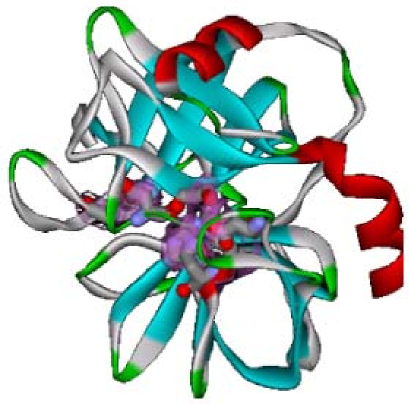
Social Network



Program Flow

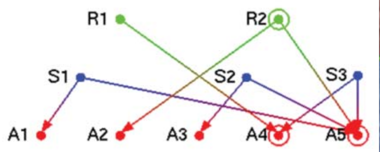


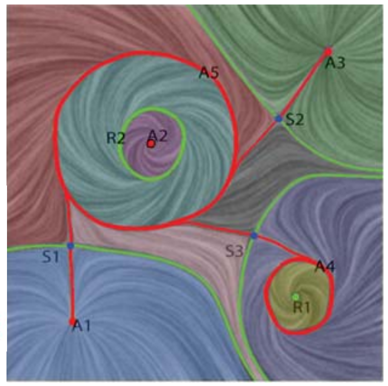
Chemical Compound

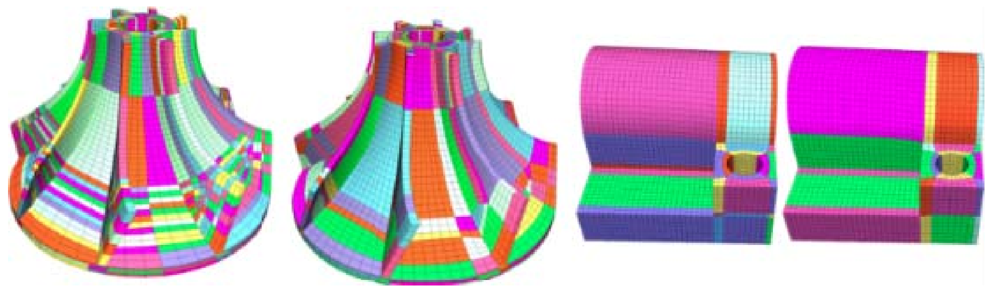


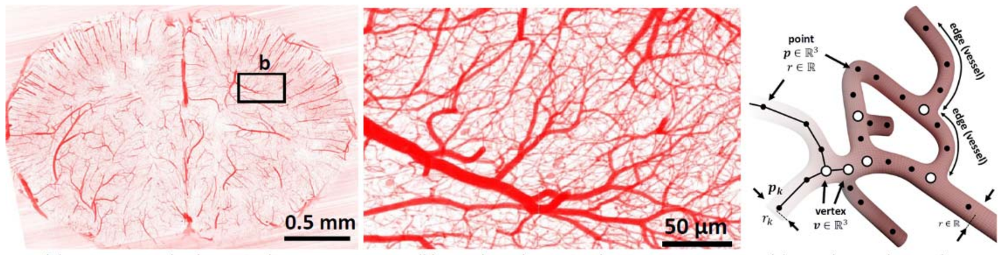
Protein Structure

Graphs are everywhere

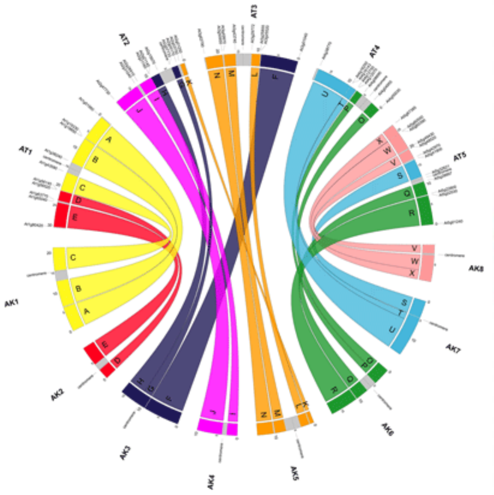


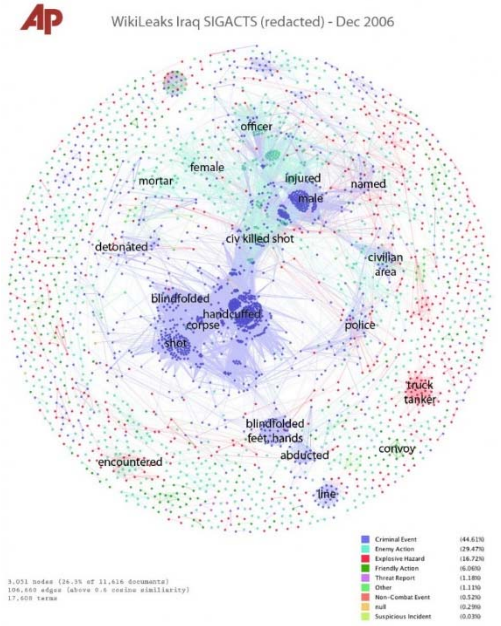






(a) mouse coronal volumetric slice (b) vascular volume visualization (c) vascular graph encoding



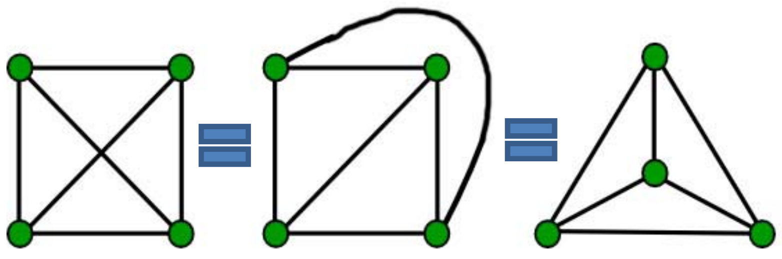


Basic Concepts

– 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*



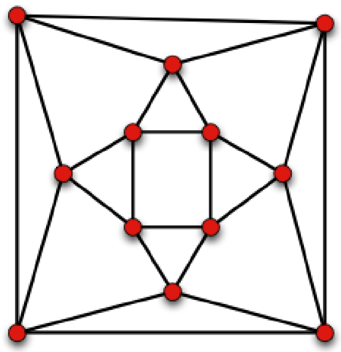


Image source: Google images

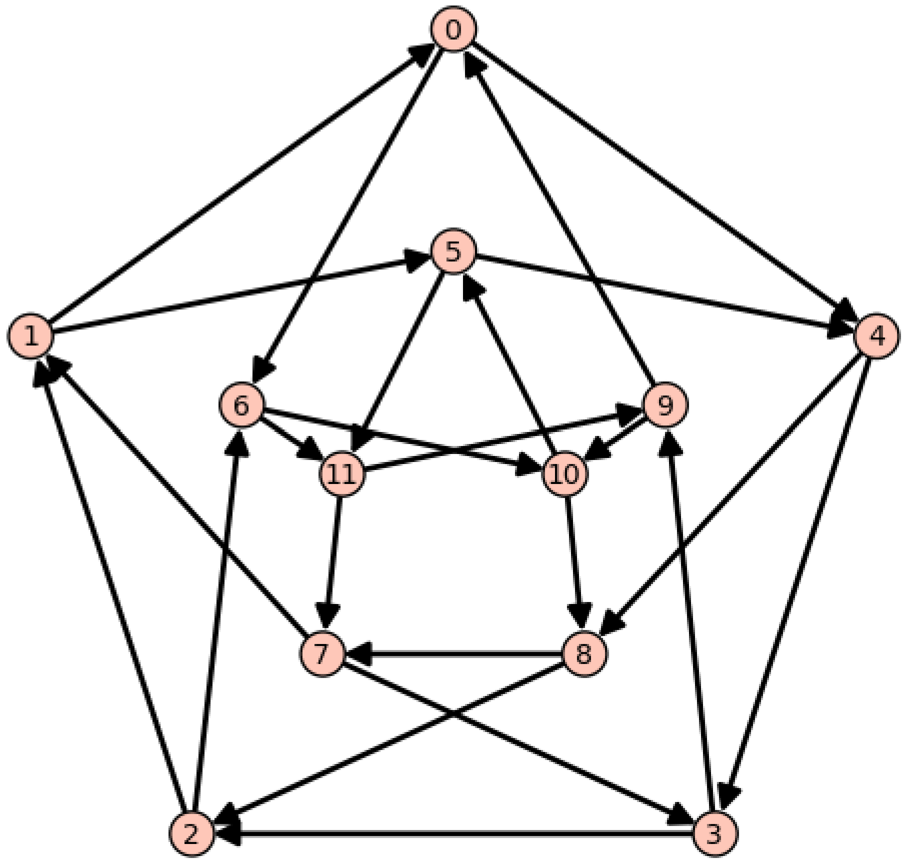
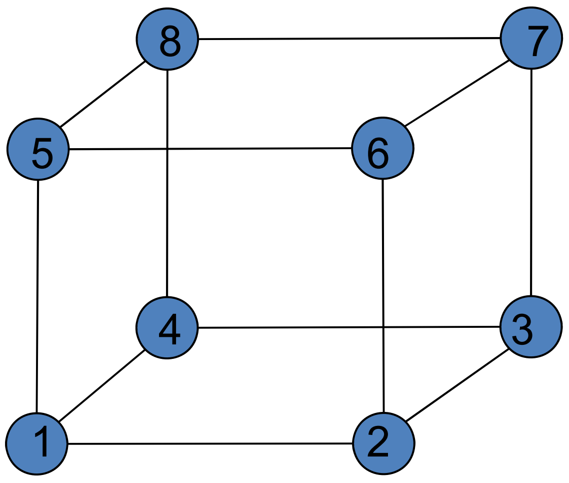


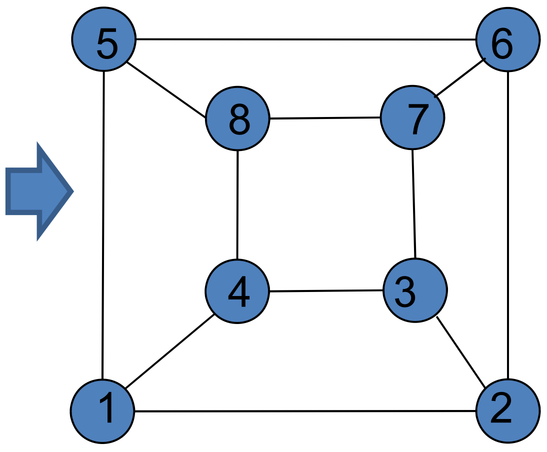
Image source: http://www.sagemath.org/doc/thematic_tutorials/linear_programming.html/

Planar graphs

Non-planar graphs



3-D



planar

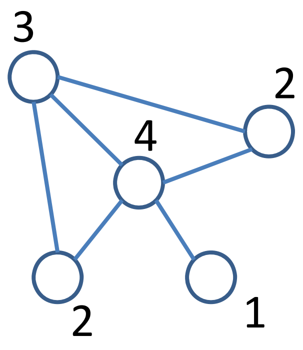
Basic Concepts

– 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

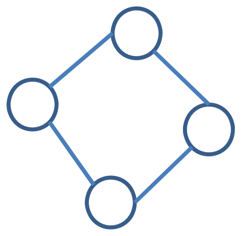
– The **degree** of a node, $\deg(v)$, is the number of edges that connect to the node

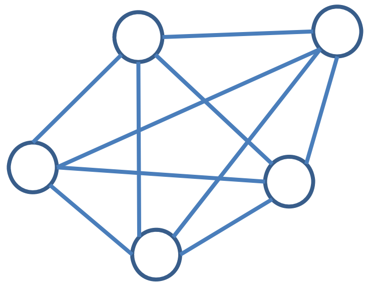


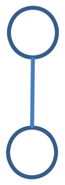
the degrees of the individual graph nodes

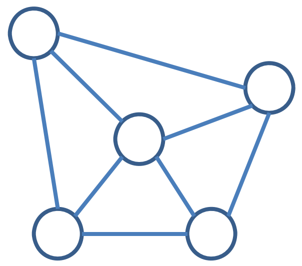
Basic Concepts

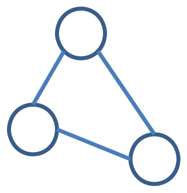
- The order of the graph G, $n = |V|$
- The size of the graph G, $m = |E|$
- A graph is planar if it can be drawn in a plane without any of the edges crossing
- The **degree** of a node, $\deg(v)$, is the number of edges that connect to the node
- The density of the graph G, $\frac{m}{\binom{n}{2}}$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



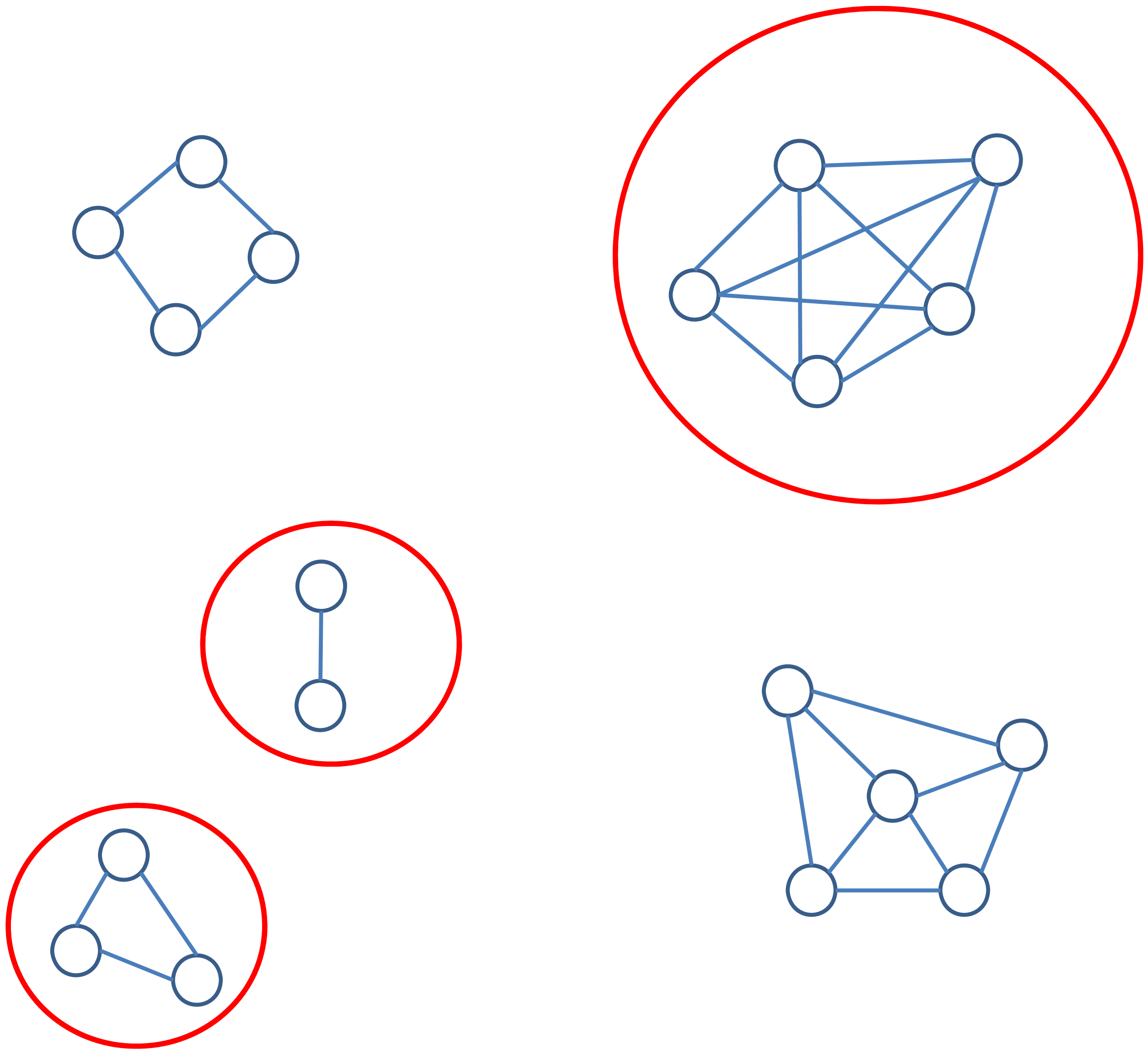






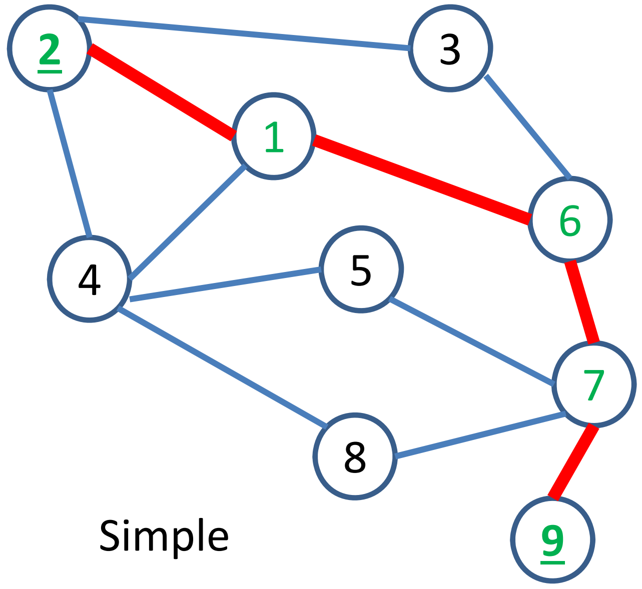


Which of these graphs are complete?



Basic Concepts (II)

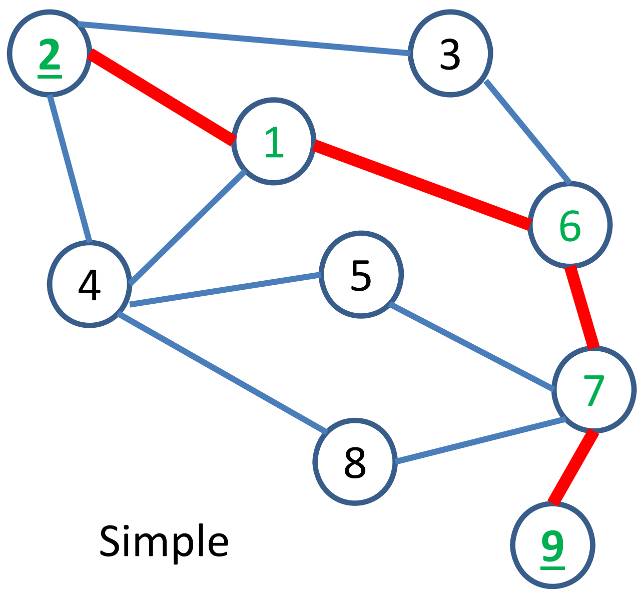
- A path from v to u in a graph $G = (V, E)$ is *a sequence of edges* in E starting at vertex $v_0 = v$ and ending at vertex $v_{k+1} = u$.
- The path is simple if no vertex is repeated



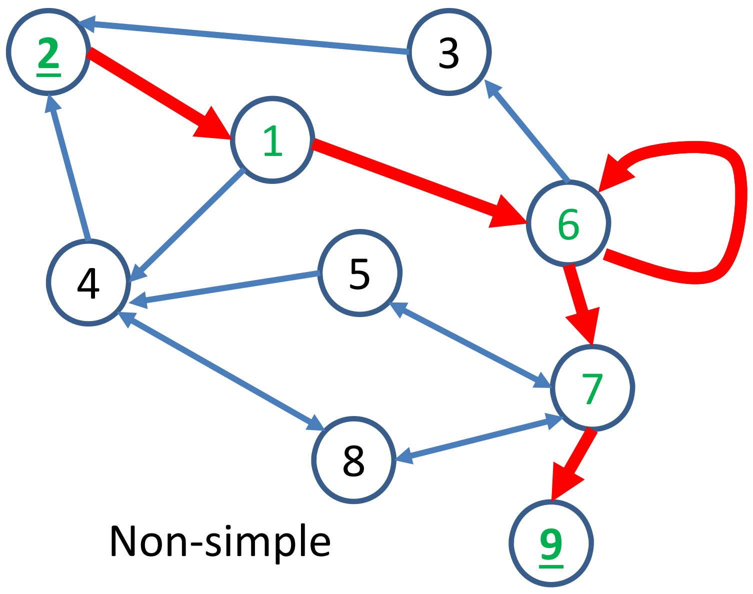
Simple

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



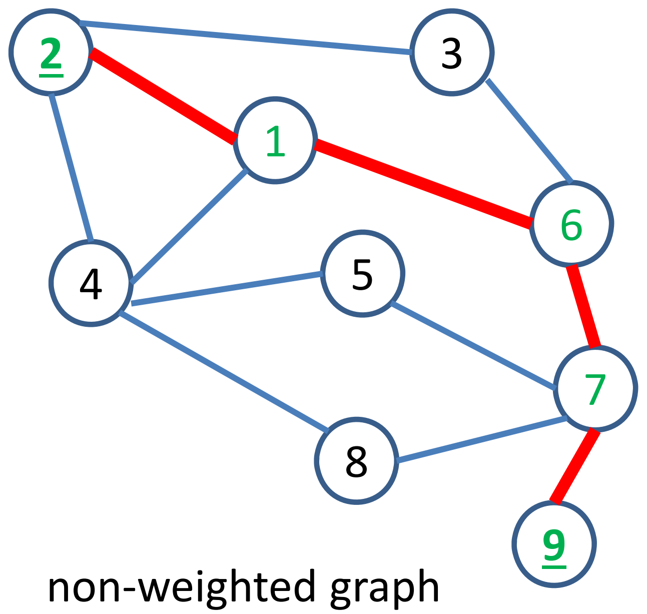
Simple



Non-simple

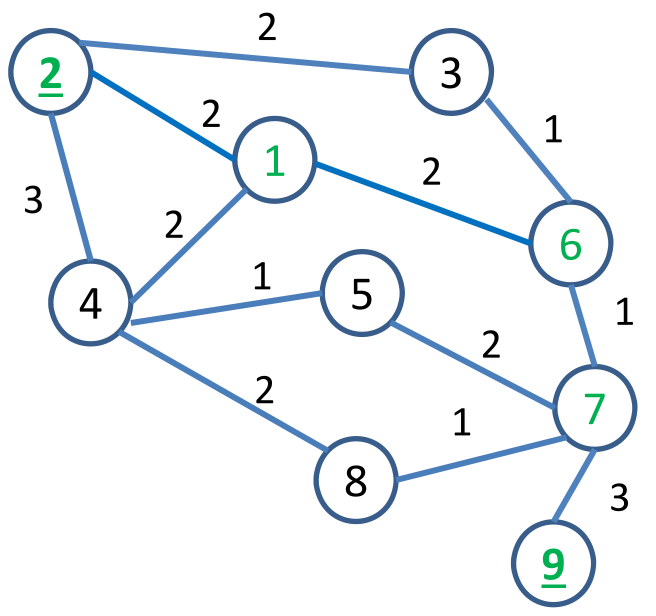
Basic Concepts (II)

- The length of the path is the number of edges on it
- The distance between two nodes is the **shortest path** connecting them.



length = 4

non-weighted graph

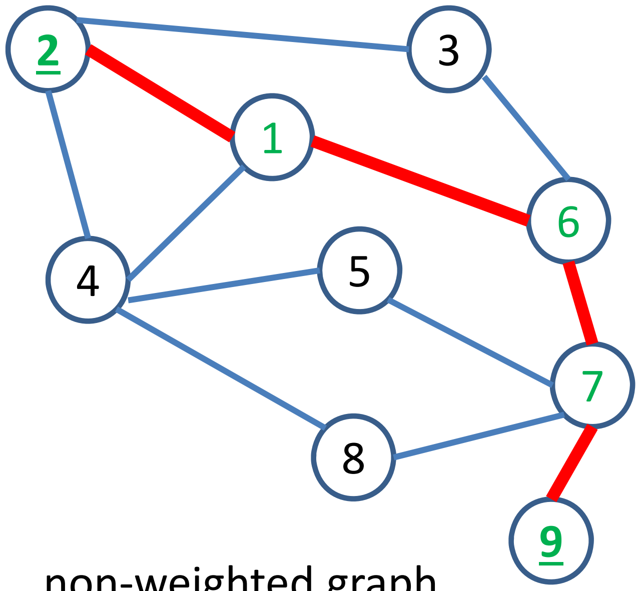


what is the shortest path between node **2** and node **9** ?

weighted graph

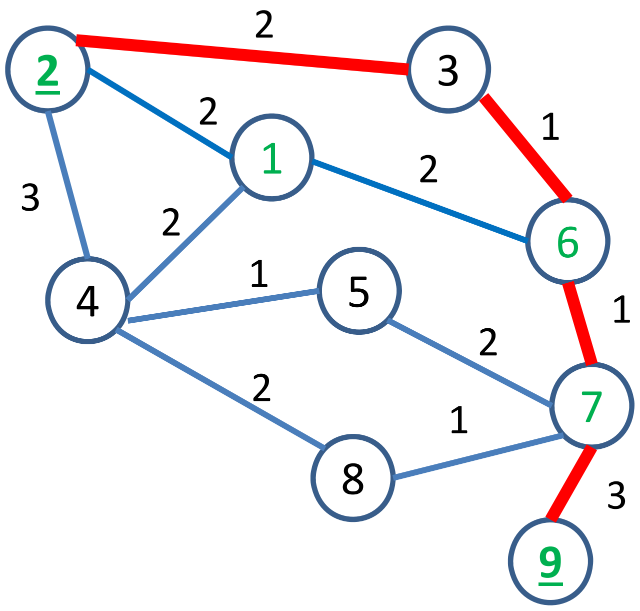
Basic Concepts (II)

- The length of the path is the number of edges on it
- The distance between two nodes is the **shortest path** connecting them.



length = 4

non-weighted graph

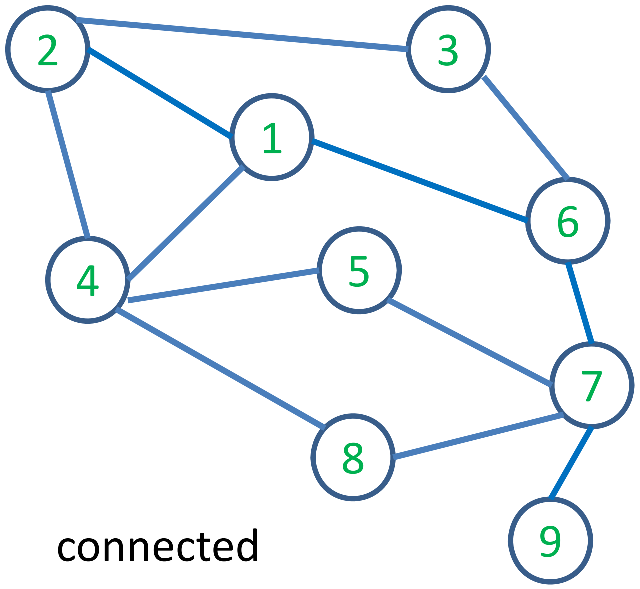


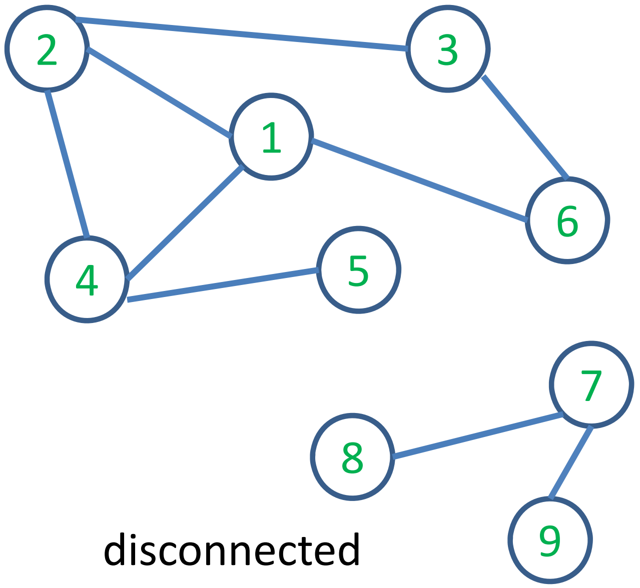
distance between node **2** and node **9** is 7

weighted graph

Basic Concepts (II)

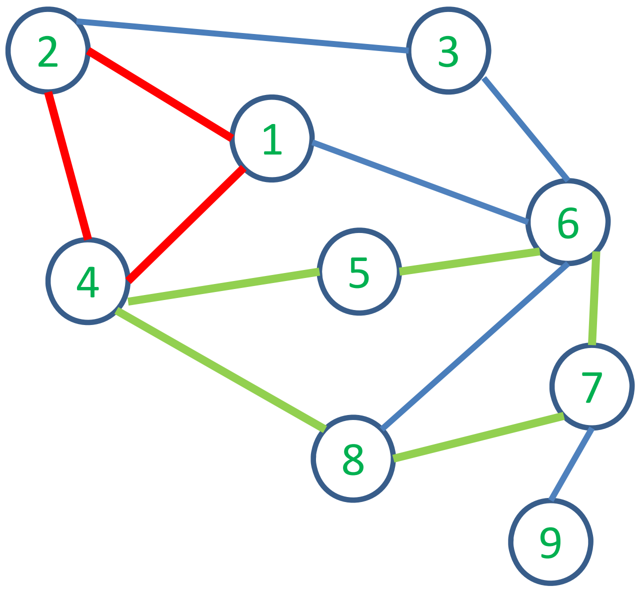
- The length of the path is the number of edges on it
- The distance between two nodes is the **shortest path** connecting them.
- A graph is <u>*connected*</u> if there exist paths between all pairs of vertices; otherwise, it is <u>*disconnected*</u>.
- The *minimum* number of edges that would need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

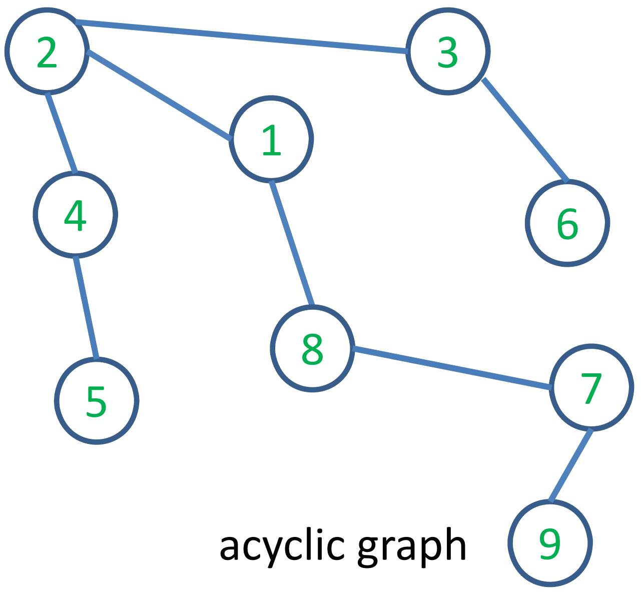




Basic Concepts (III)

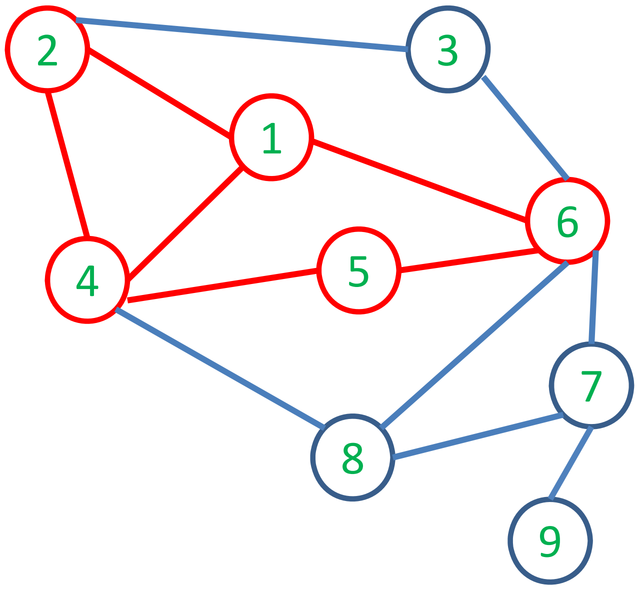
– A *cycle* is a simple path that begins and ends at the same vertex.
– A graph that contains no cycle is *acyclic* and is also called *forest*.
– A connected forest is called a ***tree***.





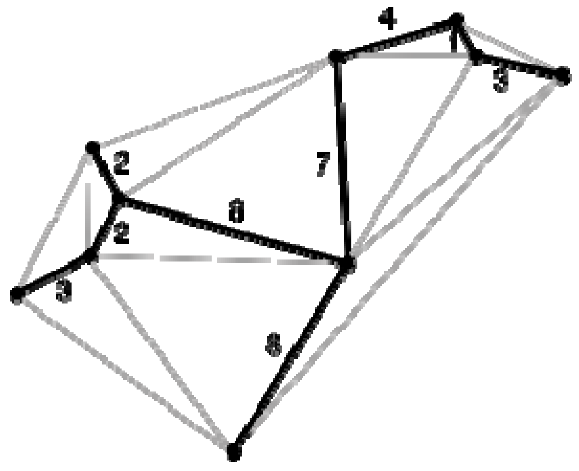
Basic Concepts (IV)

– 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.

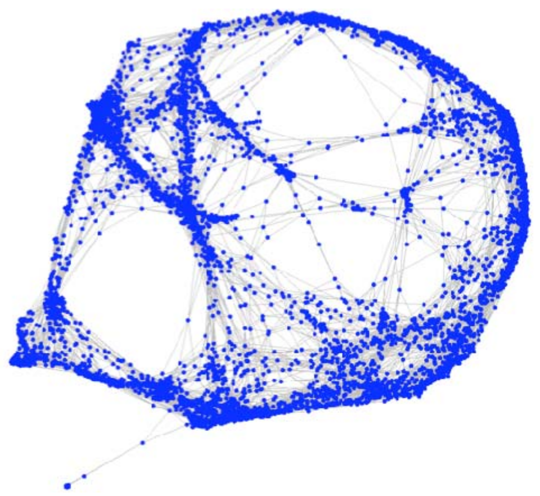


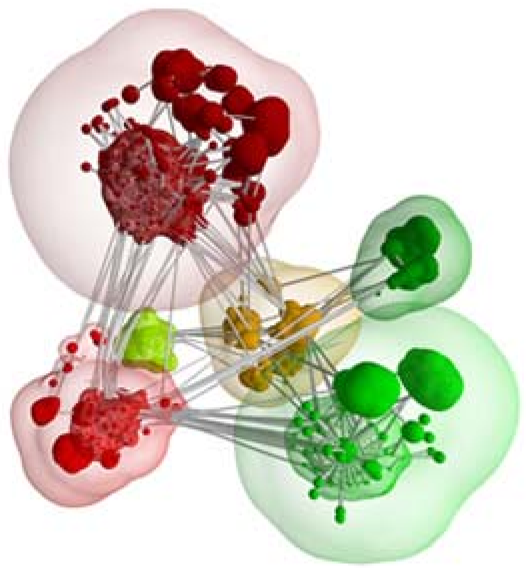
Basic Concepts (IV)

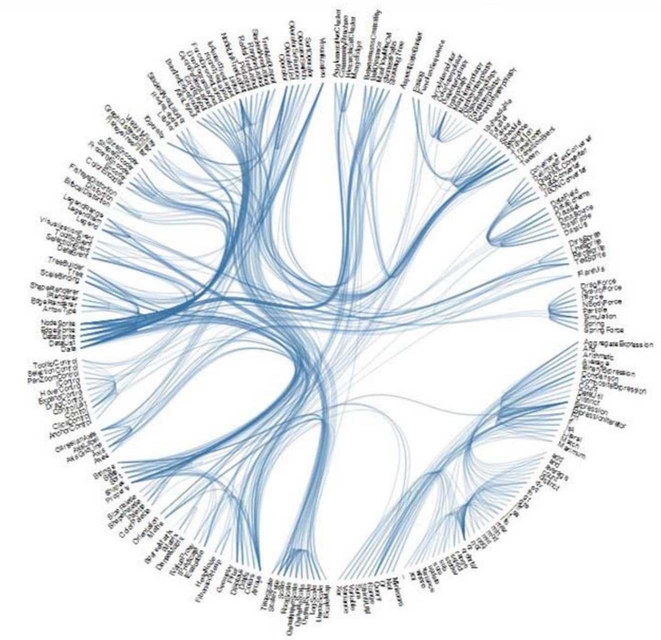
– A connected acyclic subgraph that includes all vertices in V is called a *spanning tree* of G.
  - A spanning tree has exactly $n - 1$ edges
  - If the edges have weights, the spanning tree with smallest total weights is called the *minimum spanning tree* (there may exist several of them)

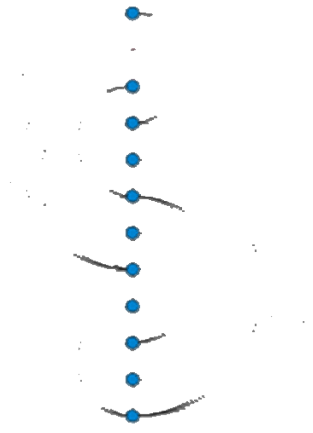


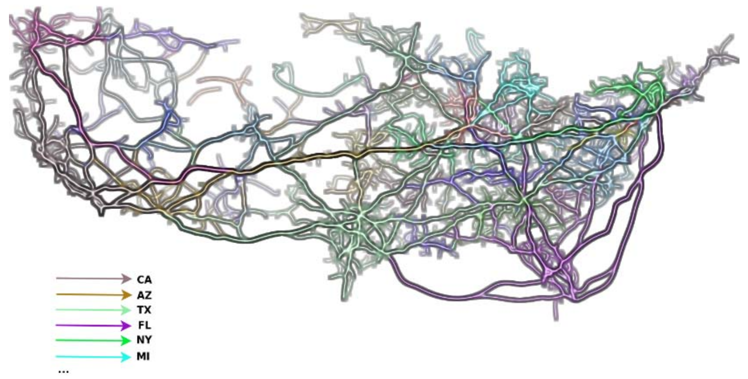
http://www.i-cherubini.it/mauro/blog/2006/04/06/minimum-spanning-tree-of-urban-tapestries-messages/

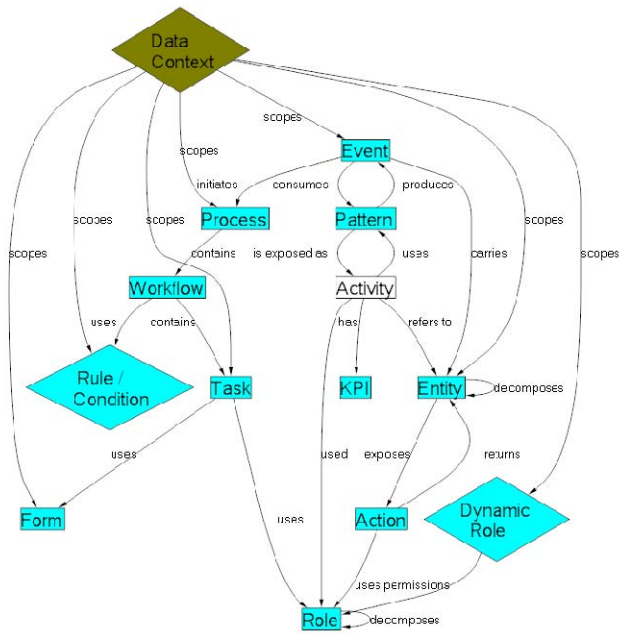












Some free graph layout tools

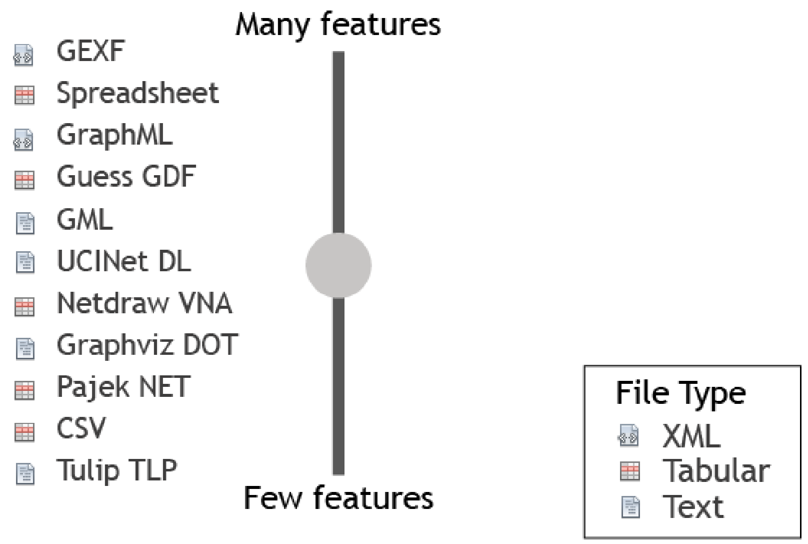
Gephi: https://gephi.org/

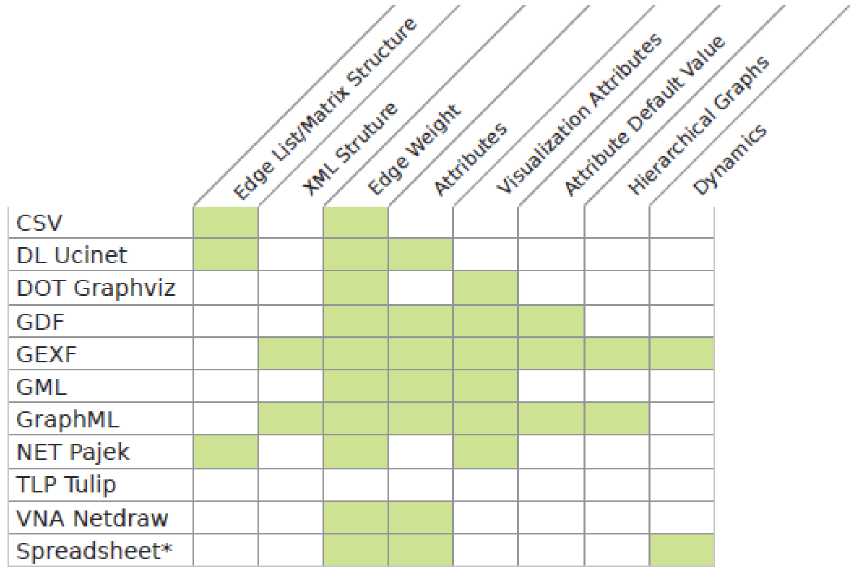
Graphviz: http://www.graphviz.org/

How is a graph represented?

Usually stored as a list of graph nodes, followed by a list of edges

File formats according to **Gephi** (https://gephi.org/users/supported-graph-formats/)





	Edge List/Matrix Structure	XML Structure	Edge Weight	Attributes	Visualization Attributes	Attribute Default Value	Hierarchical Graphs	Dynamics
CSV	🟩		🟩					
DL Ucinet	🟩		🟩	🟩				
DOT Graphviz			🟩		🟩			
GDF			🟩	🟩	🟩	🟩		
GEXF		🟩	🟩	🟩	🟩	🟩	🟩	🟩
GML			🟩	🟩	🟩			
GraphML		🟩	🟩	🟩	🟩	🟩	🟩	
NET Pajek	🟩		🟩		🟩			
TLP Tulip								
VNA Netdraw			🟩	🟩				
Spreadsheet*			🟩	🟩				🟩

Basic Graph Layout Techniques

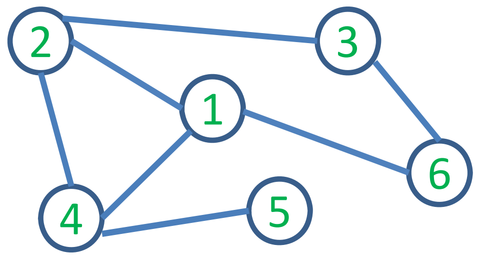
- Force-directed layout
- Arc-diagram
- Adjacency matrix
- Circular layout

Basic Graph Layout Techniques

- Force-directed layout
- Arc-diagram
- Adjacency matrix
- Circular layout

Force-Directed Layout of Graph

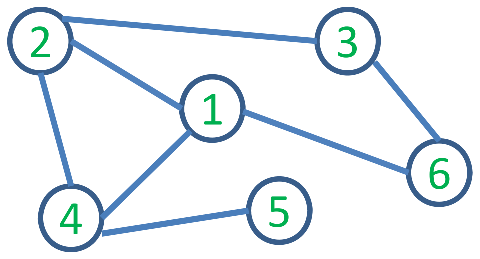
- **Presumption**:
  - The most common graphical representation of a network is a **node-link diagram**, where each node is shown as a point, circle, polygon, or some other small graphical object, and each edge is shown as a line segment or curve connecting two nodes.



Force-Directed Layout of Graph

- **Presumption**:
  - 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 close*.



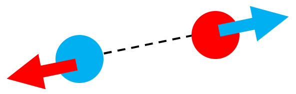
Force-Directed Layout of Graph

- Specifically, imagine that we simulate two forces: 1) **a repulsive force** between all pairs of nodes, and 2) **a spring force** between all pairs of adjacent nodes.

Force-Directed Layout of Graph

- Specifically, imagine that we simulate two forces: 1) **a repulsive force** between all pairs of nodes, and 2) 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.

Force-Directed Layout of Graph

- Specifically, imagine that we simulate two forces: 1) **a repulsive force** between all pairs of nodes, and 2) **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

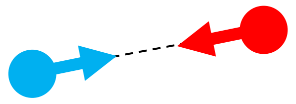
$$Fr = Kr / d^2$$

(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-Directed Layout of Graph

- Implementation
  - ***Data structure***: 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.
  - ***Algorithm:*** 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-Directed 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 =0 & i2!=i1 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 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 =0 & i2!=i1 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 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 =0 & i2!=i1 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
```

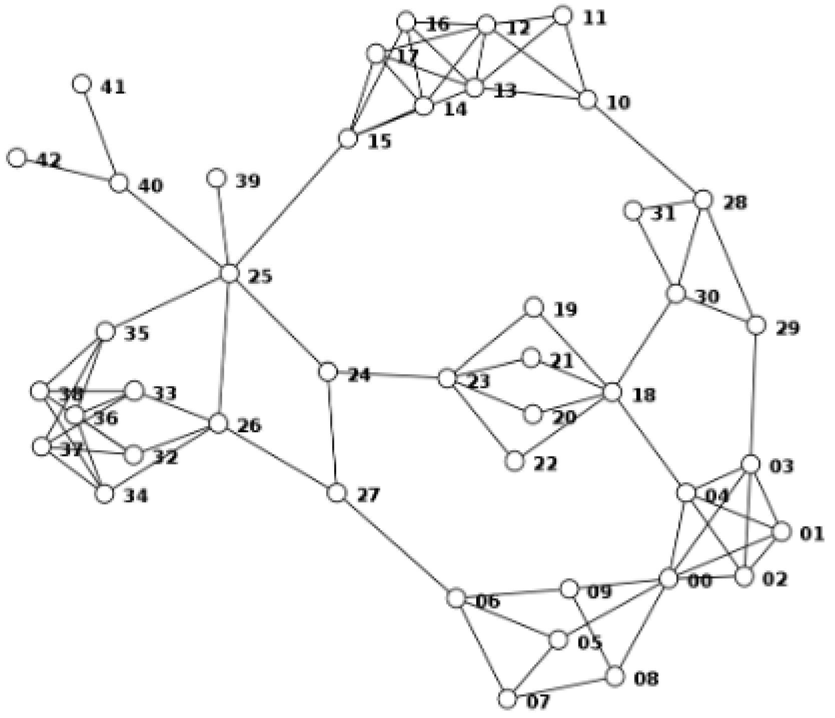
Force-Directed Layout of Graph

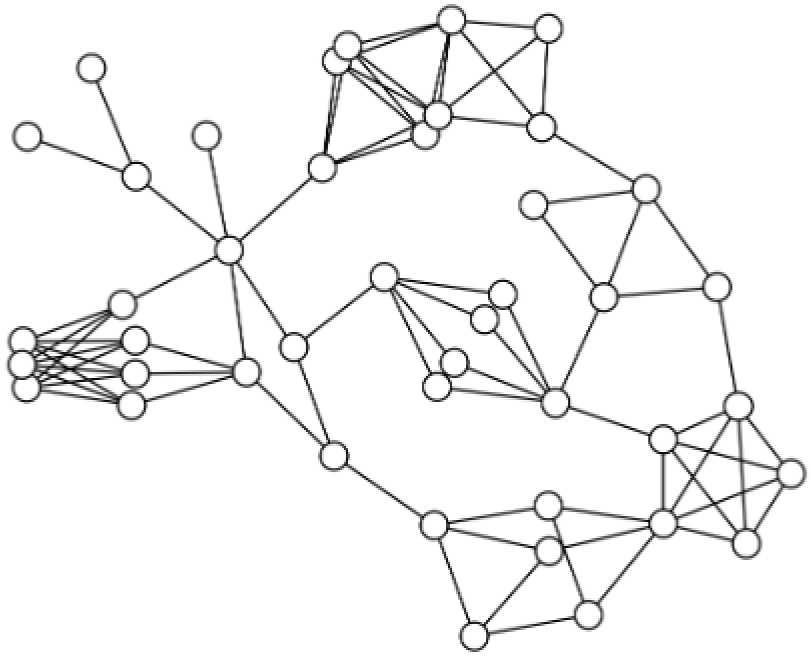
```
L = ... // spring rest length
K_r = ... // repulsive force constant
K_s = ... // spring constant
delta_t = ... // time step

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 =0 & i2!=i1 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 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 =0 & i2!=i1 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 of Graph





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

Force-Directed 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-Directed 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-Directed 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-Directed Layout of Graph

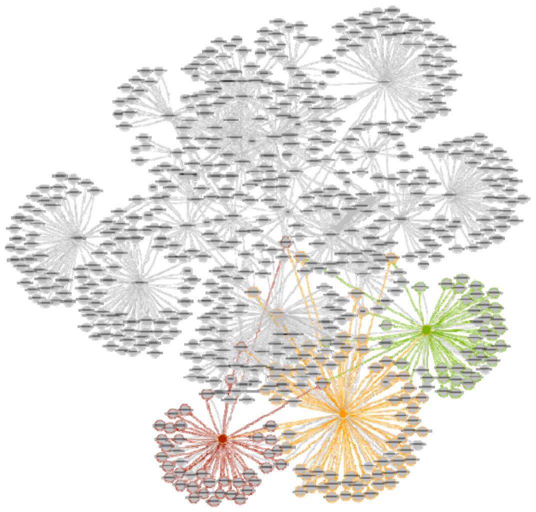
- Limitations and Improvements
  - There are infinitely many pairs of $(Kr\ , Ks)$ values that cause the layout to converge to the same final “shape” (i.e., the same angles between edges, differing only in edge lengths). A simpler user interface would allow the user to change a single parameter corresponding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

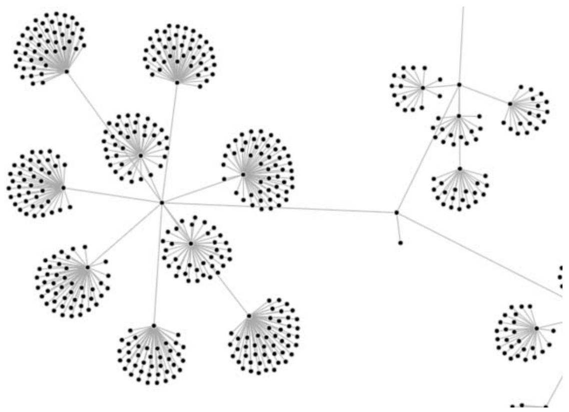
Force-Directed Layout of Graph

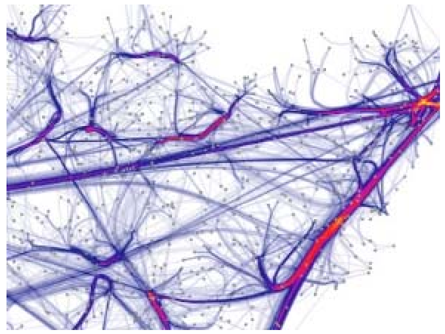
- In the pseudocode above, the computation of repulsive forces is a bottleneck, since *it requires $O(N^2)$* time, where N is the number of nodes.

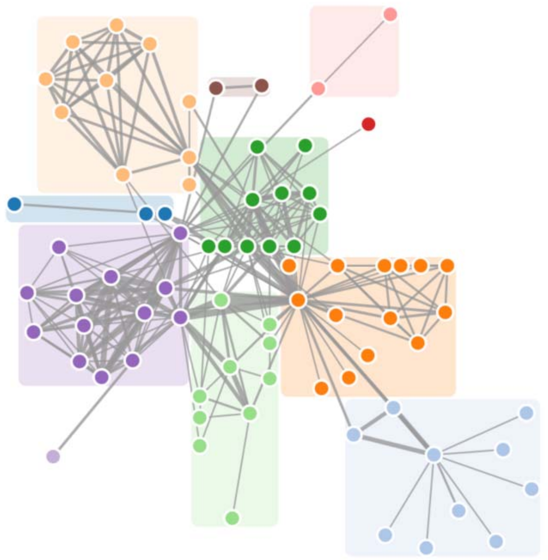
- Possible solution:
  - We could eliminate the repulsive force, and instead simulate springs of length L between all adjacent nodes, as well as springs of length $2L$ between all nodes that are two edges apart, and possibly springs of length $3L$ between nodes that are three edges apart, etc., up to 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 (low density graph)*, the computation time may be much less than $O(N^2)$.

Force-direct layout is one the most important graph layout techniques!





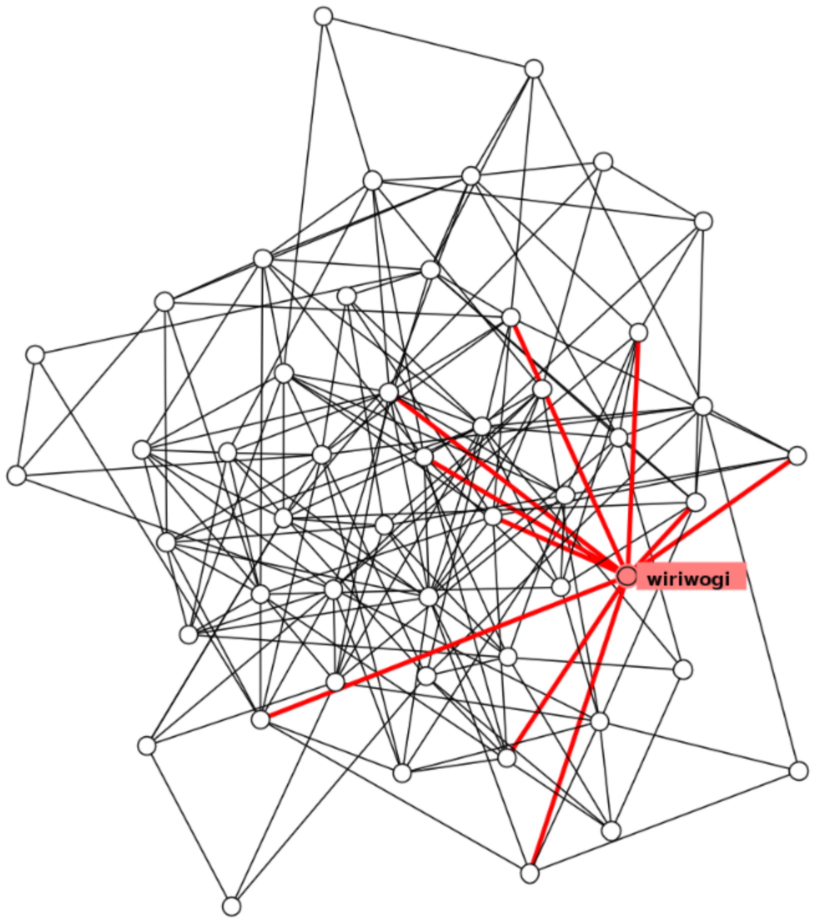




GEM, LGL, GRIP, FM^3, implemented in ***Tulip***
https://en.wikipedia.org/wiki/Tulip_(software)

Force-Directed Drawing Algorithms.
http://cs.brown.edu/people/rtamassi/gdhandbook/chapters/force-directed.pdf

Force-Directed Layout of Graph

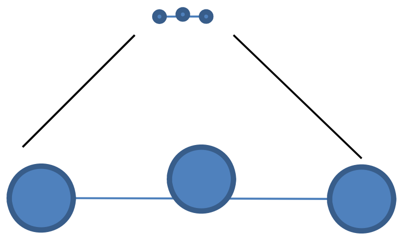


Force-directed node-link diagram of a random 50-node, 200-edge graph.

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



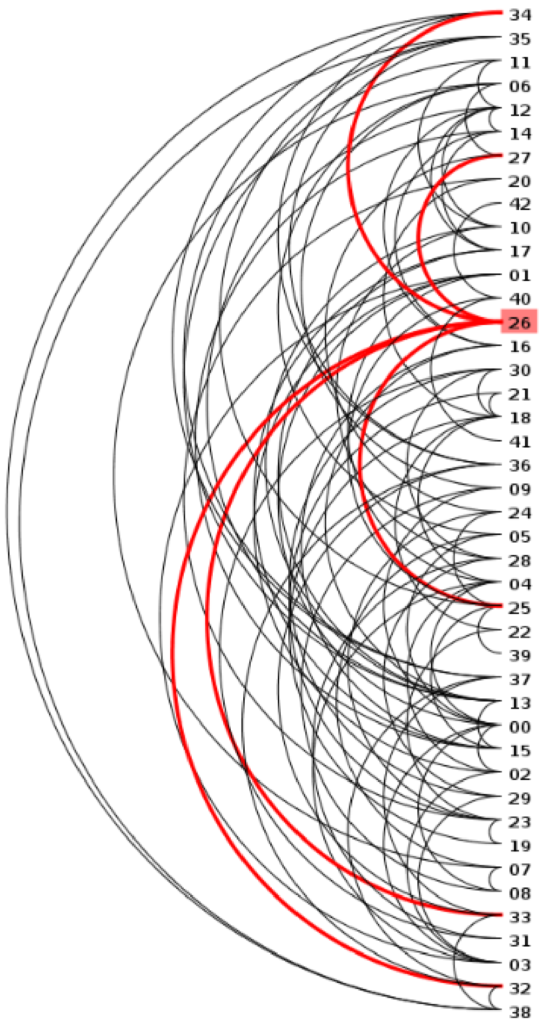
Basic Graph Layout Techniques

- Force-directed layout
- Arc-diagram
- Adjacency matrix
- 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.

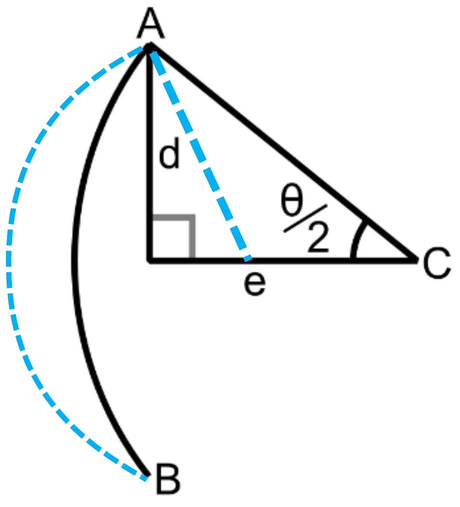


Arc diagrams of a 43-node, 80-edge network

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 $e = d/(\tan\left(\frac{\theta}{2}\right))$.

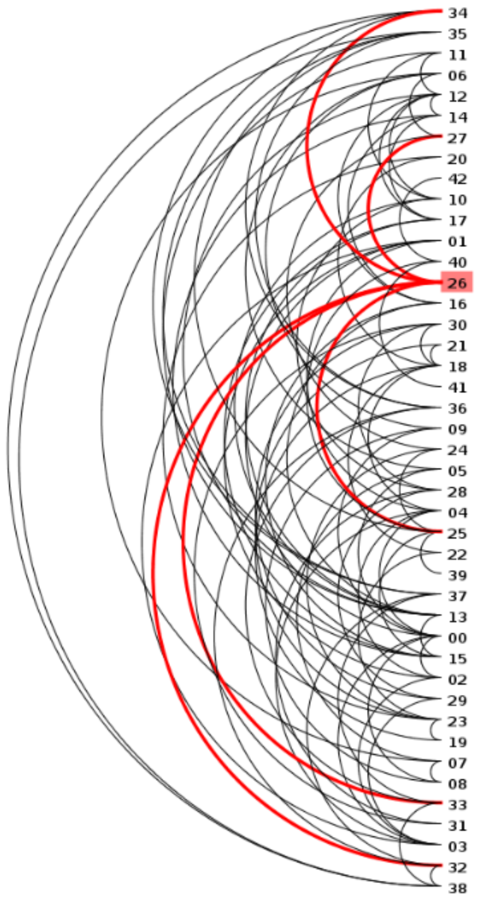


An arc covering angle θ, with center C

Arc Diagrams and Barycenter Ordering

Sorting the nodes:

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

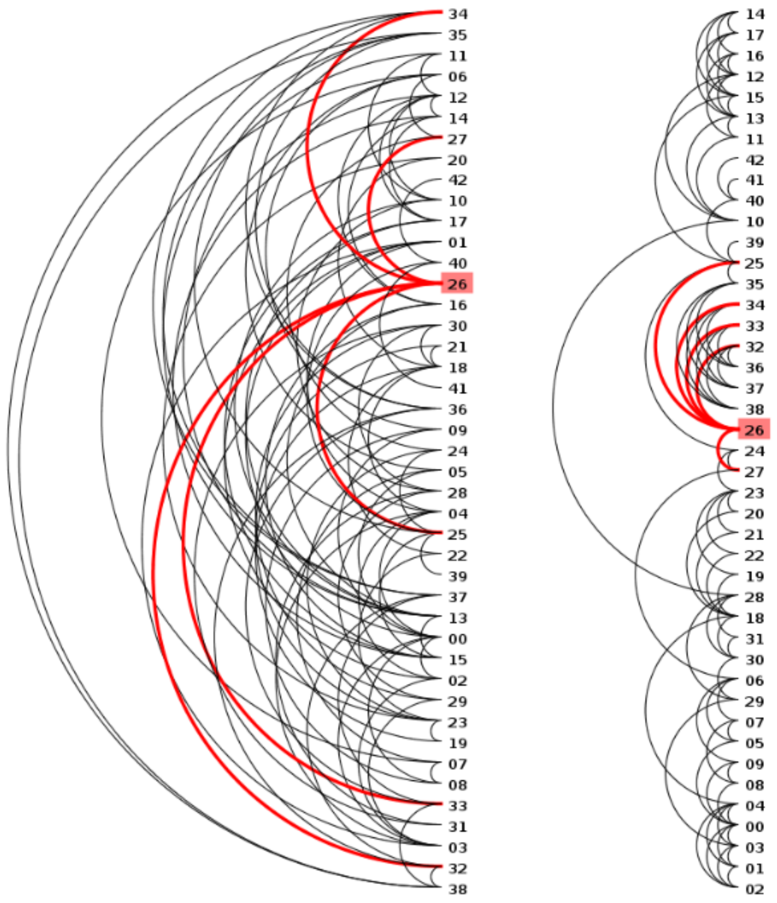


Left: with a random ordering and 180-degree arcs.

Arc Diagrams and Barycenter Ordering

Sorting the nodes:

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



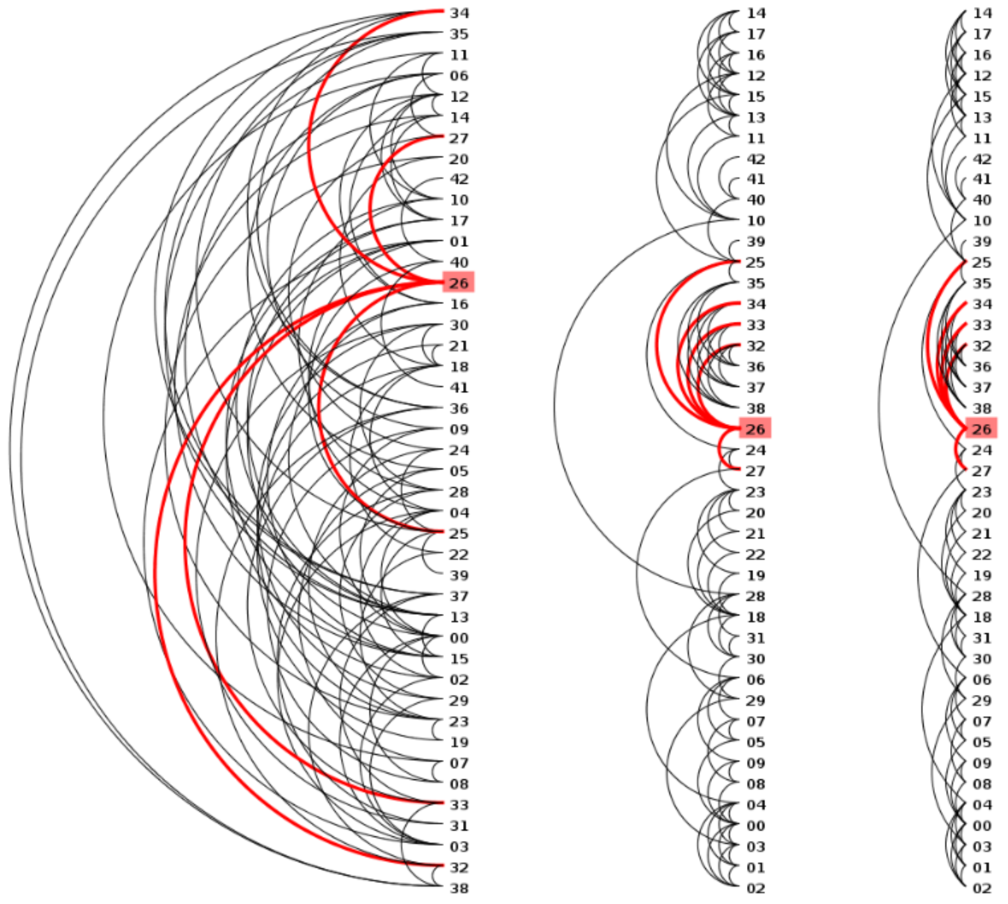
Left: with a random ordering and 180-degree arcs.
Right: after applying the barycenter heuristic to order the nodes.

Same graph, two different arc diagram layouts!

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.

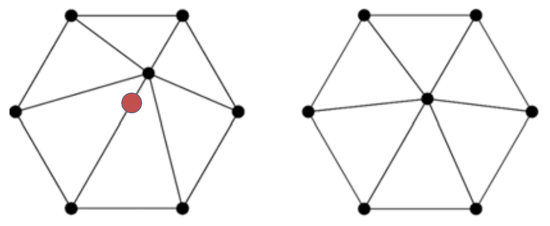
Same graph, three different arc diagram layouts!

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.

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

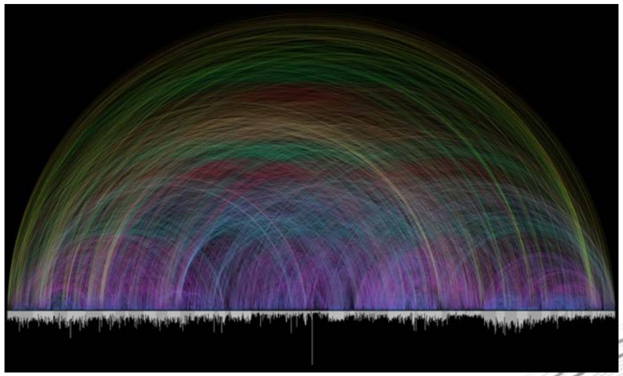
Other sorting of the nodes:

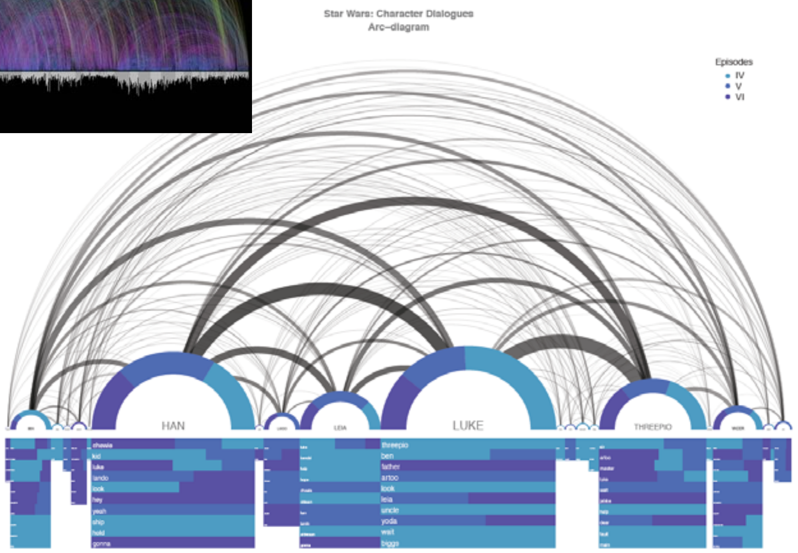
The nodes within an arc diagram might be sorted in other ways. For example, if each node has an associated label, and represents an object with a size, time-stamp, or other attribute, the nodes in the arc diagram might be sorted alphabetically, or by size, time, etc., helping the user to analyze the 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.

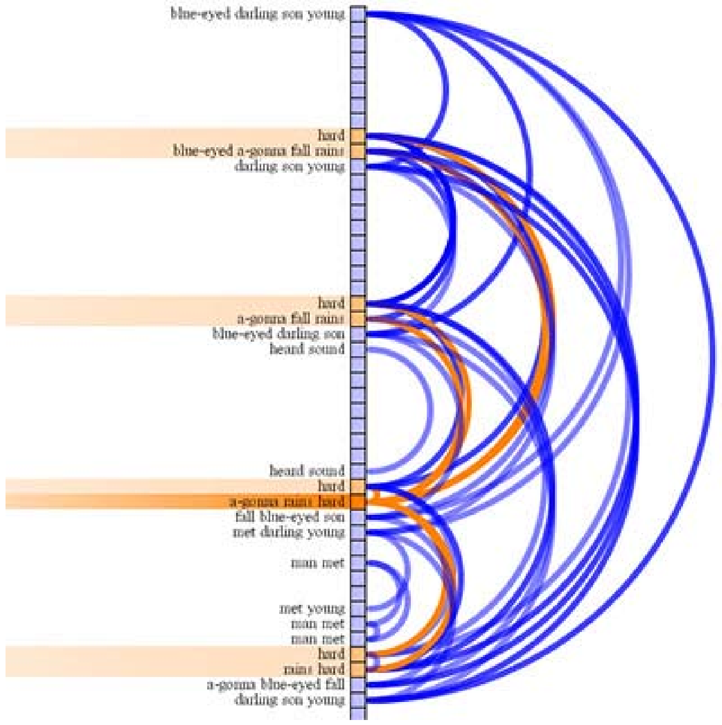
Arc Diagrams and Barycenter Ordering

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 (or left or bottom) 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.





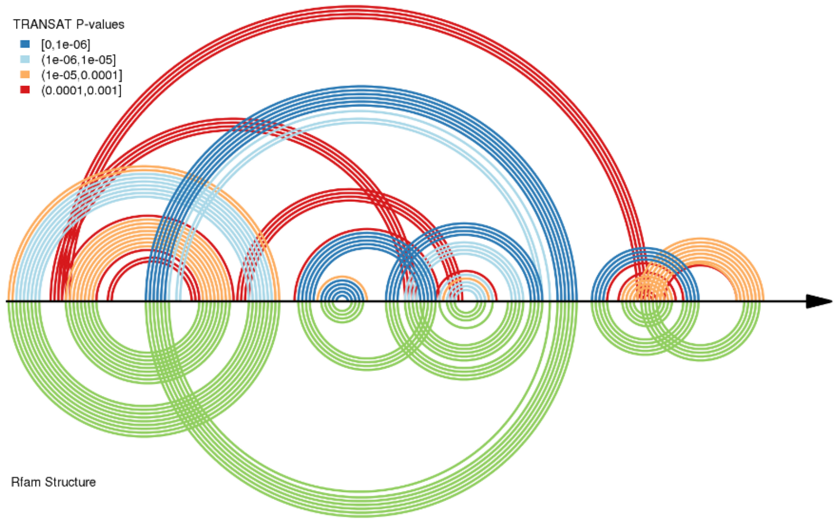


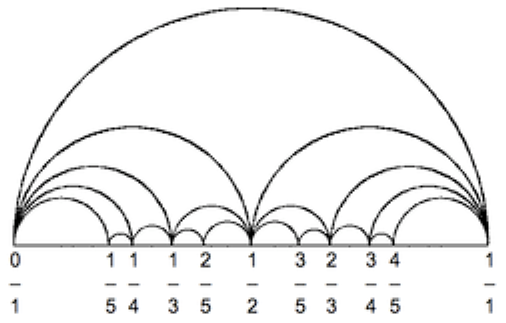
Arc Diagrams and Barycenter Ordering

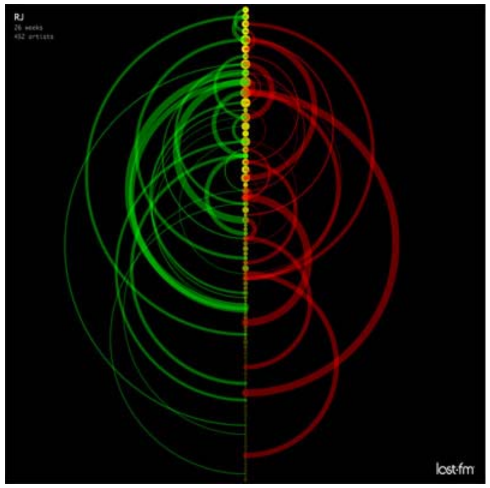
The linear arrangement of nodes in an arc diagram has many advantages.

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.

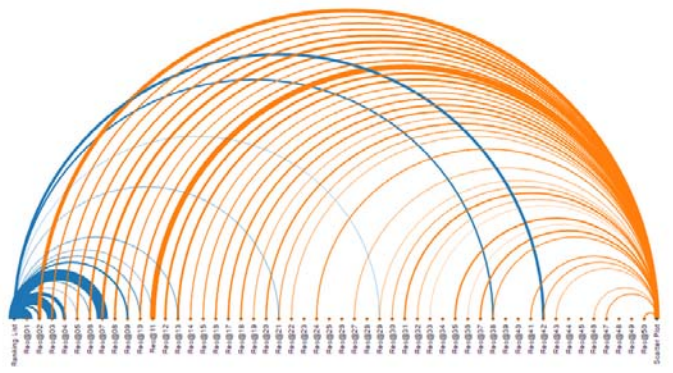
Arc diagrams can also be incorporated as an axis within a larger graphic or visualization

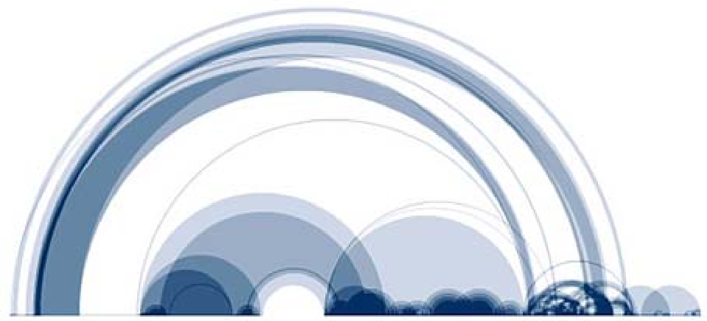


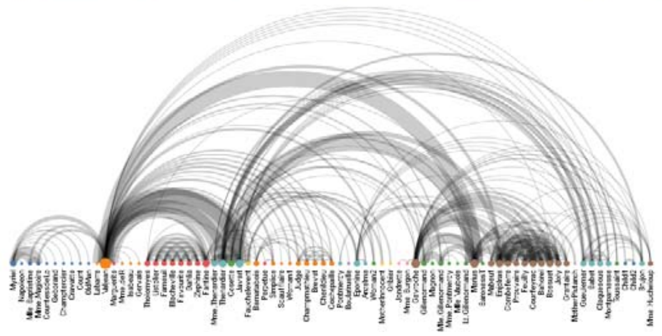




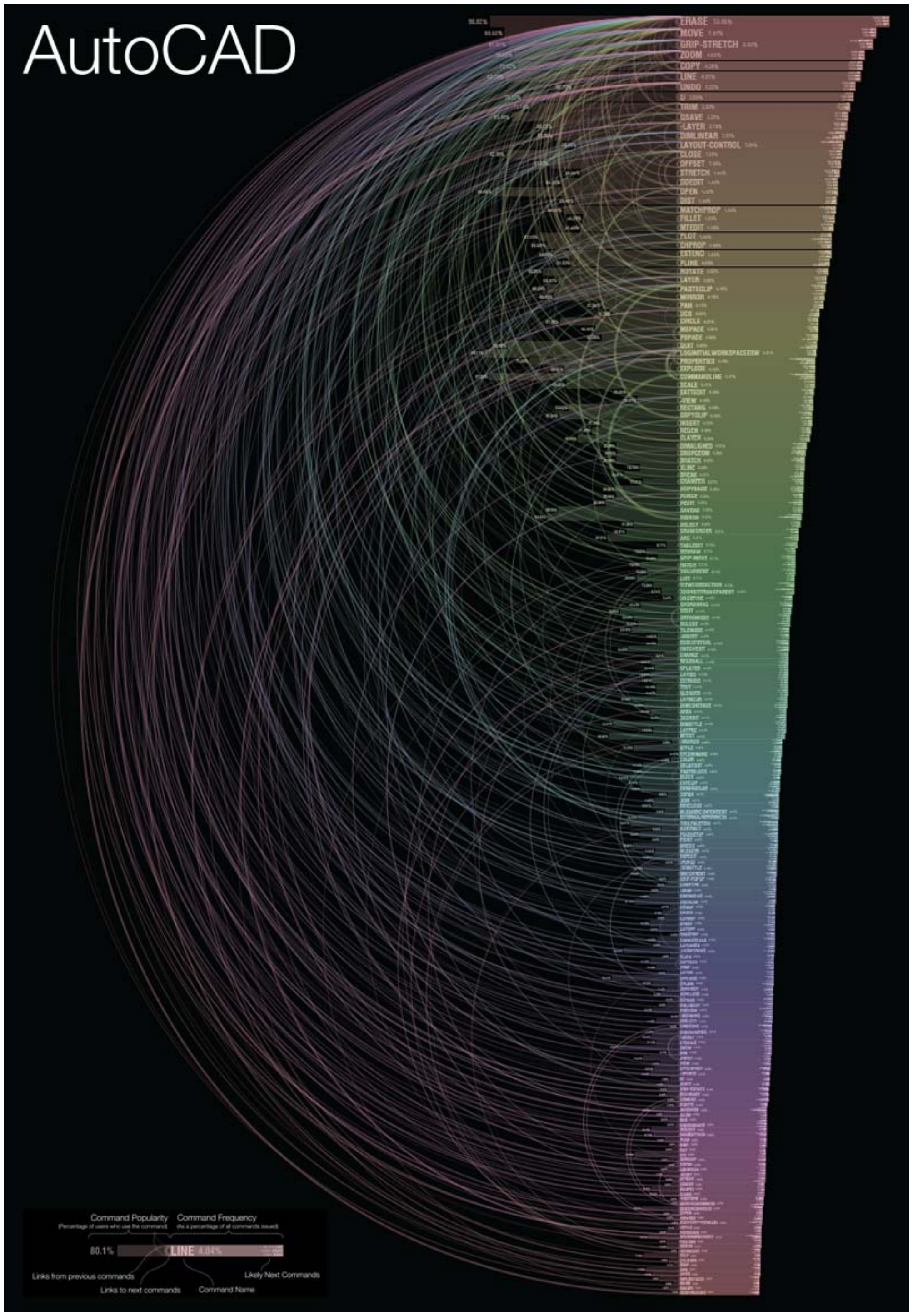
Variations of arc-diagram







https://en.wikipedia.org/wiki/Arc_diagram


AutoCAD
ERASE
MOVE
GRIP-STRETCH
ZOOM
COPY
LINE
UNDO
Command Popularity
(Percentage of users who use the command)
Command Frequency
(As a percentage of all commands issued)
LINE 4.04%
Links from previous commands
Links to next commands
Command Name
Likely Next Commands