Graph Visualization
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
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*
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, \( \text{deg}(v) \), 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 *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/
Basic Graph Layout Techniques

- Force-direct layout
- Arc-diagram
- Adjacent matrix
- Circular layout
Basic Graph Layout Techniques

- Force-direct layout
- Arc-diagram
- Adjacent matrix
- Circular layout
Force-Direct 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 near.
Force-Direct 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
  \[
  F_r = K_r / d^2
  \]
  (a definition inspired by inverse-square laws such as Coulomb’s law), where \( K_r \) is some constant.

- If the nodes are adjacent, let the spring force between them be
  \[
  F_s = K_s \left( d - L \right)
  \]
  (inspired by Hooke’s law), where \( K_s \) 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 forces `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

1 \( L = \ldots \) // spring rest length
2 \( K_r = \ldots \) // repulsive force constant
3 \( K_s = \ldots \) // spring constant
4 \( \delta_t = \ldots \) // time step
5
6 \( N = \text{nodes.length} \)
7
8 // initialize net forces
9 for \( i = 0 \) to \( N-1 \)
10 \( \text{nodes}[i].force_x = 0 \)
11 \( \text{nodes}[i].force_y = 0 \)
12
13 // repulsion between all pairs
14 for \( i1 = 0 \) to \( N-2 \)
15 \( \text{node1} = \text{nodes}[i1] \)
16 for \( i2 = 0 \) & \( i2 \neq i1 \) to \( N-1 \)
17 \( \text{node2} = \text{nodes}[i2] \)
18 \( dx = \text{node2.x - node1.x} \)
19 \( dy = \text{node2.y - node1.y} \)
20 if \( dx \neq 0 \) or \( dy \neq 0 \)
21 \( \text{distanceSquared} = dx * dx + dy * dy \)
22 \( \text{distance} = \sqrt{\text{distanceSquared}} \)
23 \( \text{force} = K_r / \text{distanceSquared} \)
24 \( fx = \text{force} * dx / \text{distance} \)
25 \( fy = \text{force} * dy / \text{distance} \)
26 \( \text{node1.force}_x = \text{node1.force}_x - fx \)
27 \( \text{node1.force}_y = \text{node1.force}_y - fy \)
28 \( \text{node2.force}_x = \text{node2.force}_x + fx \)
29 \( \text{node2.force}_y = \text{node2.force}_y + fy \)
30
31 // spring force between adjacent pairs
32 for \( i1 = 0 \) to \( N-1 \)
33 \( \text{node1} = \text{nodes}[i1] \)
34 for \( j = 0 \) to \( \text{node1.neighbors.length}-1 \)
35 \( i2 = \text{node1.neighbors}[j] \)
36 \( \text{node2} = \text{nodes}[i2] \)
37 if \( i1 < i2 \)
38 \( dx = \text{node2.x - node1.x} \)
39 \( dy = \text{node2.y - node1.y} \)
40 if \( dx \neq 0 \) or \( dy \neq 0 \)
41 \( \text{distance} = \sqrt{dx*dx + dy*dy} \)
42 \( \text{force} = K_s * (\text{distance} - L) \)
43 \( fx = \text{force} * dx / \text{distance} \)
44 \( fy = \text{force} * dy / \text{distance} \)
45 \( \text{node1.force}_x = \text{node1.force}_x + fx \)
46 \( \text{node1.force}_y = \text{node1.force}_y + fy \)
47 \( \text{node2.force}_x = \text{node2.force}_x - fx \)
48 \( \text{node2.force}_y = \text{node2.force}_y - fy \)
49
50 // update positions
51 for \( i = 0 \) to \( N-1 \)
52 \( \text{node} = \text{nodes}[i] \)
53 \( dx = \delta_t * \text{node.force}_x \)
54 \( dy = \delta_t * \text{node.force}_y \)
55 \( \text{displacementSquared} = dx*dx + dy*dy \)
56 if ( \text{displacementSquared} > \text{MAX_DISPLACEMENT_SQUARED} )
57 \( s = \sqrt{\text{MAX_DISPLACEMENT_SQUARED} / \text{displacementSquared}} \)
58 \( dx = dx * s \)
59 \( dy = dy * s \)
60 \( \text{node.x} = \text{node.x} + dx \)
61 \( \text{node.y} = \text{node.y} + dy \)
Force-Direct Layout of Graph

1 \( L = \ldots \) // spring rest length
2 \( K_r = \ldots \) // repulsive force constant
3 \( K_s = \ldots \) // spring constant
4 \( \delta_t = \ldots \) // time step
5
6 \( N = \text{nodes.length} \)

8 // initialize net forces
9 for \( i = 0 \) to \( N-1 \)
10 \( \text{nodes}[i].force\_x = 0 \)
11 \( \text{nodes}[i].force\_y = 0 \)

13 // repulsion between all pairs
14 for \( i_1 = 0 \) to \( N-2 \)
15 \( \text{node}_1 = \text{nodes}[i_1] \)
16 for \( i_2 = 0 \) to \( i_2 != i_1 \) to \( N-1 \)
17 \( \text{node}_2 = \text{nodes}[i_2] \)
18 \( \text{dx} = \text{node}_2.x - \text{node}_1.x \)
19 \( \text{dy} = \text{node}_2.y - \text{node}_1.y \)
20 if \( \text{dx} != 0 \) or \( \text{dy} != 0 \)
21 \( \text{distanceSquared} = \text{dx}^2 + \text{dy}^2 \)
22 \( \text{distance} = \sqrt{\text{distanceSquared}} \)
23 \( \text{force} = K_r / \text{distance} \)
24 \( \text{fx} = \text{force} \times \text{dx} / \text{distance} \)
25 \( \text{fy} = \text{force} \times \text{dy} / \text{distance} \)
26 \( \text{node}_1.\text{force}\_x = \text{node}_1.\text{force}\_x - \text{fx} \)
27 \( \text{node}_1.\text{force}\_y = \text{node}_1.\text{force}\_y - \text{fy} \)
28 \( \text{node}_2.\text{force}\_x = \text{node}_2.\text{force}\_x + \text{fx} \)
29 \( \text{node}_2.\text{force}\_y = \text{node}_2.\text{force}\_y + \text{fy} \)

31 // spring force between adjacent pairs
32 for \( i_1 = 0 \) to \( N-1 \)
33 \( \text{node}_1 = \text{nodes}[i_1] \)
34 for \( j = 0 \) to \( \text{node}_1.\text{neighbors}.\text{length}-1 \)
35 \( i_2 = \text{node}_1.\text{neighbors}[j] \)
36 \( \text{node}_2 = \text{nodes}[i_2] \)
37 if \( i_1 < i_2 \)
38 \( \text{dx} = \text{node}_2.x - \text{node}_1.x \)
39 \( \text{dy} = \text{node}_2.y - \text{node}_1.y \)
40 if \( \text{dx} != 0 \) or \( \text{dy} != 0 \)
41 \( \text{distance} = \sqrt{\text{dx}^2 + \text{dy}^2} \)
42 \( \text{force} = K_s \times (\text{distance} - L) \)
43 \( \text{fx} = \text{force} \times \text{dx} / \text{distance} \)
44 \( \text{fy} = \text{force} \times \text{dy} / \text{distance} \)
45 \( \text{node}_1.\text{force}\_x = \text{node}_1.\text{force}\_x + \text{fx} \)
46 \( \text{node}_1.\text{force}\_y = \text{node}_1.\text{force}\_y + \text{fy} \)
47 \( \text{node}_2.\text{force}\_x = \text{node}_2.\text{force}\_x - \text{fx} \)
48 \( \text{node}_2.\text{force}\_y = \text{node}_2.\text{force}\_y - \text{fy} \)

52 // update positions
51 for \( i = 0 \) to \( N-1 \)
52 \( \text{node} = \text{nodes}[i] \)
53 \( \text{dx} = \delta_t \times \text{node}.\text{force}\_x \)
54 \( \text{dy} = \delta_t \times \text{node}.\text{force}\_y \)
55 \( \text{displacementSquared} = \text{dx}^2 + \text{dy}^2 \)
56 if \( \text{displacementSquared} > \text{MAX\_DISPLACEMENT\_SQUARED} \)
57 \( s = \sqrt{\text{MAX\_DISPLACEMENT\_SQUARED} / \text{displacementSquared}} \)
58 \( \text{dx} = \text{dx} \times s \)
59 \( \text{dy} = \text{dy} \times s \)
60 \( \text{node}.x = \text{node}.x + \text{dx} \)
61 \( \text{node}.y = \text{node}.y + \text{dy} \)
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.
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., $\text{displacementSquared} > \text{MAX\_DISPLACEMENT\_SQUARED}$ rather than $\text{displacement} > \text{MAX\_DISPLACEMENT}$), 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 move larger distances earlier in the process, 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 $O(N^2)$.
Limitations and Improvements

As can be seen in the left example, the multiple crossings of edges can make it unclear when certain edges pass close to a node or are connected to a node. 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.
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.
Basic Graph Layout Techniques

• Force-direct layout
• Arc-diagram
• Adjacent 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 and Barycenter Ordering

To program a subroutine that draws an arc covering angle $\theta$ 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 $e = d/\left(\tan\left(\frac{\theta}{2}\right)\right)$. 

An arc covering angle $\theta$, 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.
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:

```javascript
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:

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

Lines 1 through 14 would be inside a loop that iterates several times, hopefully until convergence to a near-optimal 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 $O(N \log N)$ 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.
Variations of arc-diagram

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